TDK dolgozat

Szerzők: Bagossy Attila, Vécsi Ádám

Debreceni Egyetem

Informatikai Kar

Számítógéptudományi Tanszék

CryptID – Platformfüggetlen Identity-based Encryption megoldás

Témavezető:
Dr. Pethő Attila
egyetemi tanár

Szerzők:
Bagossy Attila, *PTI MSc, 2. évfolyam*
Vécsi Ádám, *PTI MSc, 2. évfolyam*

Lezárva: 2019. 01. 05.

Tartalomjegyzék

Köszönetnyilvánítás

Szeretnénk megköszönni a dolgozat elkészítésében nyújtott segítséget a témavezetőnknek, Dr. Pethő Attilának. Az elmúlt hónapokban megannyi személyes egyeztetés és levélváltás során látott el minket nélkülözhetetlen szakmai tanácsokkal.

A kutatást a „Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein” (EFOP-3.6.3-VEKOP-16-2017-00002) című projekt támogatta. A projekt a Magyar Kormány és az Európai Szociális Alap társfinanszírozásában valósult meg.

1. fejezet

Bevezetés

A dolgozatunkban az általunk fejlesztett CryptID Identity-based Encryption (IBE) könyvtárat mutatjuk be. Ez a Boneh-Franklin IBE egy újszerű implementációja, mely reményeink szerint nemcsak egy újabb megvalósítás, hanem valóban olyan jellemzőkkel bír, melyek a már rendelkezésre álló megoldások versenytársává avathatják. Ehhez természetesen olyan sajátosságok szükségesek, melyek megkülönböztetik a többi könyvtártól, mi több, bizonyos összevetésben azok elé helyezik. Úgy gondoljuk, hogy a platformfüggetlen működés, a strukturált publikus kulcs és a fejlesztő-orientált interfész ilyenek lehetnek.

1.1. Identity-based Encryption

A nyilvános kulcsú kriptográfia egy fiatal ága az IBE, melynek ötletét Adi Shamir fogalmazta meg 1984-ben. Egy olyan sémát írt le, melyben nincsen szükség sem a nyilvános, sem a titkos kulcsok előzetes cseréjére vagy nyilvántartására: a nyilvános kulcsok egyértelmű, mindenki által ismert azonosítók (például egy telefonszám), míg a titkos kulcsokat egy megbízható harmadik fél, a Private Key Generator (PKG) hozza létre. Ekképpen az IBE nem igényel a Public Key Infrastructure-höz (PKI) hasonló rendszert a kulcsok kezeléséhez, hiszen az adatok titkosításához szükséges kulcsokat a rendszer minden résztvevője ismeri, a visszafejtéshez pedig egyetlen féllel, a PKG-vel kell kapcsolatba lépni. Habár utóbbi aggályos lehet, hiszen a felhasználóknak nincsen befolyása a titkos kulcs előállítására, azonban ezt a feladatot a PKI esetén is egy külső fél, a *registration authority* (RA) végzi (Buchmann, Karatsiolis & Wiesmaier, 2013).

Az IBE gyakorlatban is használható első leírását Boneh és Franklin adta 2001-ben. Ez a rendszer azonban csak az első volt a sorban: napjainkig számos különböző IBE-rendszer született meg, melyek rendre eltérő jellemzőkkel rendelkeznek. Az elméleti előrelépéseket követte a gyakorlat is, hiszen a fejlesztők mára több implementáció közül is választhatnak.

1.2. Motiváció

Az internetre csatlakozó mobil eszközök robbanásszerű elterjedése igényt ébresztett hatékony és hordozható kriptográfiai rutinok iránt. Ugyanakkor a jelenlegi megvalósítások egyáltalán nem, vagy csak kevésbé veszik figyelembe ezt az igényt. Emiatt a CryptID kifejezetten a platformfüggetlenséget szem előtt tartva készült, legyen szó asztali, mobil vagy IoT eszközökről.

A publikus kulcsban elhelyezhető metaadatok ötlete már Boneh és Franklin cikkében is megjelenik (2001). Az elképzelés lényege, hogy az azonosítón felül további adatokat, például egy dátumot is elhelyezünk a publikus kulcsban. Az elképzelést annyira előremutatónak találtuk, hogy a CryptID ehhez teljes mértékű támogatást biztosít – a publikus kulcsok ugyanis JSON objektumokként reprezentálhatók.

Mi motiválta viszont a fejlesztő-orientált interfész létrehozását? Az ismert könyvtárakat áttekintve azt vettük észre, hogy bár nagyszerű kriptográfiai képességekkel rendelkeznek, azonban helyes működtetésükhöz számottevő matematikai és kriptográfiai háttér szükséges. Annak érdekében, hogy az IBE-t több fejlesztő tudja helyesen integrálni, a Google Tinkkel[1] azonos mottót választva egy olyan könyvtár elkészítését tűztük ki célul, melyet könnyű jól használni és nehéz (vagy nehezebb) rosszul.

1.3. A dolgozat felépítése

A dolgozatunk első három fejezete bevezető jellegű: előbb az elliptikus görbe kriptográfián, valamint az IBE-n keresztül a matematikai alapokat tárgyaljuk, majd a WebAssembly formájában a technológiai hátteret ismertetjük. Ezt követi a dolgozat fő eredményét jelentő CryptID könyvtár rétegről rétegre történő részletes bemutatása, kiemelt figyelmet szentelve a teljesítmény elemzésének. Zárásként két alkalmazás szerepel, melyek a CryptID nagyobb léptékű programokba történő integrálását demonstrálják.

[1] `https://github.com/google/tink`

2. fejezet

Elliptikus görbe kriptográfia

A fejezet célja, hogy bemutassa az Identity-based Encryption (IBE) megértéséhez szükséges matematikai hátteret. Ismerteti az elliptikus görbék matematikai elméletének felhasználását és szerepét a kriptográfiában. A fejezet elkészítéséhez Hankerson és munkatársai művét vettük alapul (2004).

2.1. Elliptikus görbék

Az elliptikus görbékkel kapcsolatban rengeteg forrás létezik, hiszen már évszázadok óta tanulmányozzák matematikusok. Ahogyan azt Lang írta: „*It is possible to write endlessly on Elliptic Curves (This is not a threat).*" (1978). Ezzel szemben dolgozatunkban egy lényegre törő rövid áttekintést szeretnénk adni.

Egy $\mathbb{F}$ test feletti elliptikus görbét azok a $P = (x, y) \in \mathbb{F}^2$ pontok alkotnak, amelyek kielégítik az alábbi egyenletet:

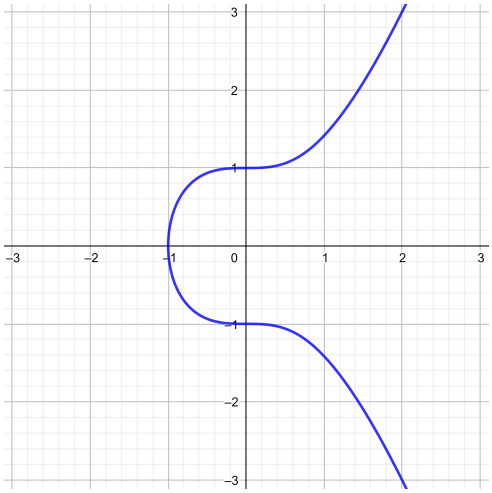
$$y^2 + axy + by = x^3 + cx^2 + dx + e,$$

ahol $a, b, c, d, e, x, y \in \mathbb{F}$ és a, b, c, d, e adottak. Az elliptikus görbék algebrai elméletéhez szükségünk van egy olyan képzetes, végtelen távoli O pontra, amely rajta van minden függőleges egyenesen és az x-tengelyre vonatkozó tükörgépe önmaga, tehát $O = -O$.

A kriptográfiában kiemelt szereppel bírnak az úgynevezett Weierstrass elliptikus görbék (Koblitz, 1987), melyek egyenlete egyszerűbb az előbb látottnál:

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

Egy ilyen görbéről látható példa a 2.1. ábrán. A dolgozat további részében is ilyen görbéket fogunk tekinteni.



2.1. ábra. Az $y^2 = x^3 + 1$ görbe a valós számok teste felett.

2.2. Műveletek elliptikus görbék pontjaival

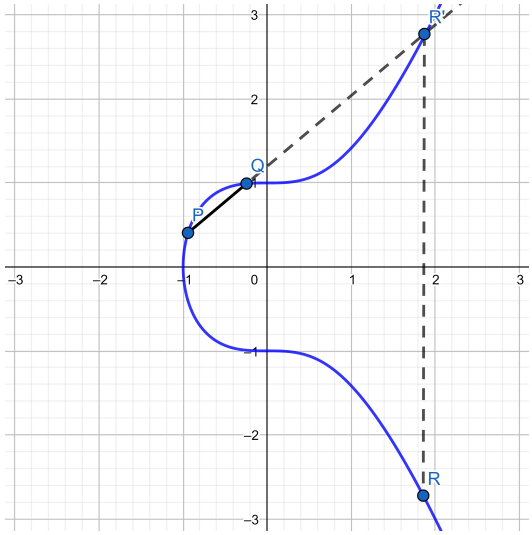
Az elliptikus görbék egy fontos tulajdonsága, hogy a görbe pontjai a megfelelően definiált összeadás művelettel Abel-csoportot alkotnak, amelyben az egységelem a korábban definiált O végtelen távoli pont.

2.2.1. Két pont összeadása

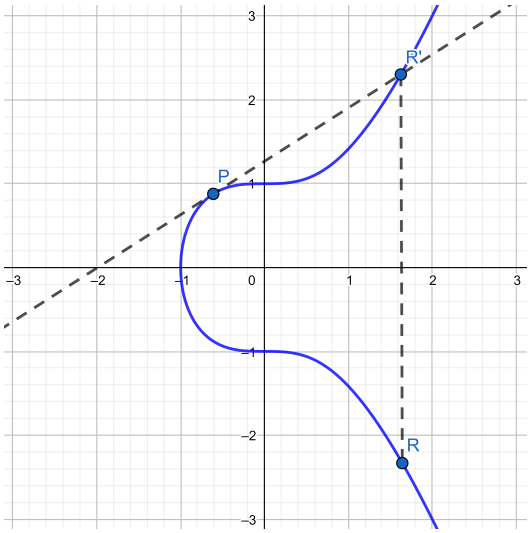
Az összeadás szabályát geometriai úton a következőképp tekintjük. Legyen E egy elliptikus görbe egy K test felett, melynek jelölése $E(K)$. Legyen $P = (x_1, y_1)$ és $Q = (x_2, y_2)$ az E elliptikus görbének két különböző pontja. Ekkor a P és Q pontok R összegét a következő módon kapjuk.

Először húzzuk be a P-re és Q-ra illeszkedő egyenest. Ez az egyenes metszeni fogja az elliptikus görbét egy harmadik pontban. Ha ezt a harmadik pontot tükrözzük az x-tengelyre, akkor megkapjuk R-t. Adott pont önmagával vett összeadása (azaz duplázása) annyiban tér el az előzőtől, hogy egy olyan egyenest szükséges felrajzolni első lépésként, amely az elliptikus görbét az adott pontban érinti.

Az előbbiekben leírt módszereket ábrázolják a 2.2a és 2.2b ábrák.



(a) Elliptikus görbe két különböző pontjának összeadása.



(b) Elliptikus görbe pontjának összeadása önmagával.

Az $E(K) : y^2 = x^3 + ax + b$ görbe tulajdonságai [1]

Egységelem. $P + O = O + P = P$, minden $P \in E(K)$ esetén.

Ellentettek. Ha $P = (x, y) \in E(K)$, akkor $(x, y) + (x, -y) = O$. Az $(x, -y)$ pontot $-P$-vel jelöljük és P ellentettjének nevezzük. $-P$ is $E(K)$ egy pontja.

Pontok összeadása. Legyen $P = (x_1, y_1) \in E(K)$ és $Q = (x_2, y_2) \in E(K)$, úgy hogy $P \neq \pm Q$. Ekkor $P + Q = (x_3, y_3)$, ahol

$$x_3 = \left(\tfrac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \text{ és } y_3 = \tfrac{y_2 - y_1}{x_2 - x_1}(x_1 - x_3) - y_1.$$

Pont duplázás. Legyen $P = (x_1, y_1) \in E(K)$, úgy hogy $P \neq -P$. Ekkor $2P = (x_3, y_3)$, ahol

$$x_3 = \left(\tfrac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \text{ és } y_3 = \tfrac{3x_1^2 + a}{2y_1}(x_1 - x_3) - y_1.$$

Ha $P = -P$, akkor $2P = O$.

2.2.2. Pontok skalár szorzása

Az elliptikus görbe pontjait tetszőleges k skalárral megszorozhatjuk, ami annyit jelent, hogy a pont k példányát összeadjuk. Tehát ezen szorzás kiszámításának triviális módja az, ha elvégzünk $k - 1$ darab összeadást, azonban ez nagy k esetén egy nagyon költséges lépéssorozat. Ezzel szemben létezik néhány kevésbé egyszerű, de sokkal hatékonyabb algoritmus is a szorzás elvégzésére.

[1] Ha K karakterisztikája nem 2.

Double-and-Add

A Horner séma (Horner, 1819) vagy *Double-and-Add* nevű módszer egy hatékony megoldást nyújt elliptikus görbe pontjának skaláris szorzására, hasonlóan polinomok Horner-elrendezéséhez: Legyen a k bináris felírása $\sum_{i=0}^{t-1} k_i 2^i$, ekkor

$$kP = \sum_{i=0}^{t-1} k_i 2^i(P) = k_0 P + 2(k_1 P + 2(k_2 P + 2(... + 2(k_{t-1} P)...))).$$

A fenti képlet pszeudokódját az Algoritmus 1. jeleníti meg.

Algoritmus 1 Double-and-Add algoritmus

```
procedure ScalarMultiply(k, P)          ▷ k = (k_{t-1}, ..., k_1, k_0)_2, P ∈ E(F_q).
    Q ← O
    for i ← t − 1 downto 0 do
        Q ← 2Q
        if k_i = 1 then
            Q ← Q + P
        end if
    end for
    return Q
end procedure
```

Ezen felül számos lehetőség van optimalizálásra, mint például újabb, gyorsabb algoritmusok használata, azonban a dolgozat szemponjából ezek nem fontosak.

2.3. Elliptikus görbék a kriptográfiában

A kriptográfiában olyan elliptikus görbéket szokás alkalmazni, amelyek véges test felett vannak definiálva. A megelőzőleg bevezetett műveletek ekkor is érvényesek, az eredmény pedig mindig a görbe egy pontja lesz. Elterjedtek az $\mathbb{F}_p$ felett értelmezett görbék, ahol p egy prímszám. Ekkor a pontok koordinátáival mindig modulo p kell számolni.

Az elliptikus görbén alapuló kriptográfiai sémák biztonságosságát az elliptikus diszkrét logaritmus probléma nehézsége adja.

Definíció. *Az **elliptikus diszkrét logaritmus probléma** (ECDLP): Adott egy E elliptikus görbe az $\mathbb{F}_q$ véges test felett, egy n rendű $P \in E(\mathbb{F}_q)$ pont, valamint egy Q pont, amely P többszöröse. Keressük azt az $l \in [0, n-1]$ egész számot, amelyre $Q = lP$ teljesül. Ezt a számot a Q pont P alapú elliptikus diszkrét logaritmusának nevezzük.*

A definícióban bevezetett diszkrét logaritmus meghatározására nem ismert hatékony algoritmus.

2.3.1. Miért válasszuk az elliptikus görbe kriptográfiát?

Napjainkban az egyik legelterjedtebb aszimmetrikus titkosítási módszer az RSA algoritmus, ami a biztonságosságát a faktorizáció problémájából nyeri. Az RSA matematikai háttere sokkal egyszerűbbnek mondható, mint az elliptikus görbéken alapuló algoritmusoké, ezért implementálása sokkal egyszerűbb.

A faktorizáció ugyanakkor kevésbé nehéz probléma, mint az elliptikus diszkrét logaritmus probléma. Ahhoz, hogy az RSA algoritmus elérje a kellő biztonságosságot, sokkal nagyobb bithosszúságú kulcsokkal kell dolgoznia, mint az elliptikus görbéken alapuló algoritmusoknak (Miller, 1985). Ahogy a NIST ajánlásából olvasható, 160-521 bithosszúságú kulcs megfelelő az elliptikus görbén alapuló algoritmusok esetén (*Digital Signature Standard (DSS)*, 2013), ezzel szemben az RSA esetén sokkal hosszabb, 1024-15360 bithosszúságú kulcsok szükségesek. Az NSA táblázat formájában is összevetette a NIST ajánlásait a különböző méretű AES kulcsok titkosításához szükséges RSA és elliptikus görbe kulcshosszakról (*The Case for Elliptic Curve Cryptography*, 2009).

A nagy kulcsok lassítják a számítást, ezért úgy gondoljuk, hosszú távon az elliptikus görbén alapuló algoritmusok jobban skálázhatók, mint az RSA.

2.3.2. Elliptikus görbe könyvtárak

Munkánk során több különböző elliptikus görbe aritmetikát megvalósító könyvtárat is megvizsgáltunk, amelyeket a 2.1. táblázat foglal össze.

Könyvtár neve	**Link**	**Licenc**
libecc	`https://github.com/ANSSI-FR/libecc`	BSD és GPL v2
MIRACL	`https://github.com/miracl/MIRACL`	GNU AGPL v3
PARI	`https://pari.math.u-bordeaux.fr/`	GNU GPL
SageMath	`http://www.sagemath.org/`	GNU GPL
snowshoe	`https://github.com/catid/snowshoe`	BSD 3-Clause

2.1. táblázat. Elliptikus görbe aritmetikát megvalósító könyvtárak.

A libecc és a snowshoe dokumentációját elégtelennek találtuk, ami rendkívül megnehezítette volna a felhasználásukat.

A SageMath bár jól dokumentált, azonban egy óriási méretű könyvtár, melynek csak néhány elemére lett volna szükségünk.

A PARI (*PARI/GP version* `2.11.0`, 2018) és a MIRACL között a döntést hozó tényező az volt, hogy a PARI fejlesztőit a témavezetőnk útján közelebbről is ismerjük.

Később azonban a fejlesztés során kellett rájönnünk, hogy céljaink elérése érdekében saját elliptikus görbe aritmetikát kell implementálnunk.

Az egyik indok, ami erre a döntésre juttatott minket, hogy böngészőben kliensoldalon, akár mobil eszközökről is használhatóvá akartuk tenni a CryptID-nek nevezett könyvtárunkat, ami megköveteli a kis méretet az internetes adatforgalom csökkentése érdekében. A vizsgált elliptikus görbe műveleteket megvalósító könyvtárak számos olyan funkciót is tartalmaznak, amikre nekünk nem volt szükségünk, ezzel fölösleges mérettöbletet alkotva.

Másik fontos szempontunk az volt, hogy a saját implementációval úgy gondoljuk, leegyszerűsítettük a jövőbeli optimalizációs és kutatási tevékenységeink folytatását ezen a területen.

3. fejezet

Identity-based Encryption

Az Identity-based Encryption olyan nyilvános kulcsú titkosítási eljárás, amely esetén a publikus kulcs egy tetszőleges karaktersorozat lehet, amely egyértelműen azonosítani tud egy entitást.

A fejezetben kifejtjük az IBE célját és működését, azonban ezt megelőzően még egy rövid betekintést adunk a párosítások működésébe és fontosságába.

3.1. Párosítás-alapú kriptográfia

A párosítás lényege, hogy egy bizonyos csoporton definiált nehéz probléma átalakítható egy könnyebb problémává egy másik csoport felett. Ez a leképezés számos új kriptográfiai séma létrejöttét tette lehetővé, köztük az Identity-based Encryptionét.

3.1.1. A párosítás elterjedése a kriptográfiában

A párosítás-alapú kriptográfia egy nagyon fiatal terület, mely a párosítások kriptoanalízisben való alkalmazásából fejlődött ki (Menezes, Vanstone & Okamoto, 1991). A MOV redukcióval sikerült szuperszinguláris görbék esetén az elliptikus görbe diszkrét logaritmus problémát redukálni egy véges testen értelmezett diszkrét logaritmus problémává.

A következő lépéseket a párosítás-alapú kriptográfia kialakulása felé Joux tette, aki a Weil és Tate párosításokat használta a Diffie-Hellmann protokoll egy variációjának létrehozására (2000).

Ezt követően készítette el Boneh és Franklin a csoportok közti bilineáris leképezésen (például Weil és Tate párosításon) alapuló IBE rendszerüket (2001).

Ezeket a munkákat tekinthetjük a párosítás-alapú kriptográfia úttörőinek.

3.1.2. A párosítás tulajdonságai

Az alfejezetben El Mrabet és Joye munkáját vesszük alapul a párosítás tulajdonságainak ismertetéséhez (2016).

Jelöljön G_1, G_2 additív, míg $G_\mathbb{T}$ multiplikatív r rendű csoportokat. Az e párosítás egy olyan $e : G_1 \times G_2 \to G_\mathbb{T}$ leképezés, amely a következő tulajdonágokkal rendelkezik:

Bilineáris. Jelölje $\mathbb{Z}_r$ az egész számok halmazát modulo r, ekkor $\forall P_1 \in G_1, P_2 \in G_2$ és $a, b \in \mathbb{Z}_r$ esetén $e(aP_1, bP_2) = e(P_1, P_2)^{ab}$.

Nem elfajuló. Ha $P_1 \neq 0_{G_1}$ és $P_2 \neq 0_{G_2}$, akkor $e(P_1, P_2) \neq 1_{G_\mathbb{T}}$, ahol 0_{G_1} (illetve 0_{G_2} és $1_{G_\mathbb{T}}$) az egységeleme a G_1 csoportnak (illetve G_2 és $G_\mathbb{T}$ csoportnak).

Hatékonyan számítható.

Nehezen megfordítható.

A kriptográfiában elterjedten használt párosítási módszerek a Weil és a Tate párosítás.

3.2. Személyre szabott titkosítás

Ahogy azt a fejezet elején említettük, az IBE egy olyan nyilvános kulcsú titkosítási eljárás, melynek esetén a publikus kulcs tetszőleges olyan karaktersorozat lehet, amely egyértelműen azonosítani tud egy entitást. Fontos azonban, hogy nemcsak az azonosító, hanem az annak hatókörét jelentő domain is tetszőleges. Lehet csupán néhány fős (például egy vállalat), vagy akár globális kiterjedésű is.

A kitalált séma célja az volt, hogy az entitások kulcscsere nélkül tudjanak egymásnak titkosított üzenetet küldeni (Shamir, 1985). Azonban éveken át sikertelenül próbáltak létrehozni jól működő IBE sémákat.

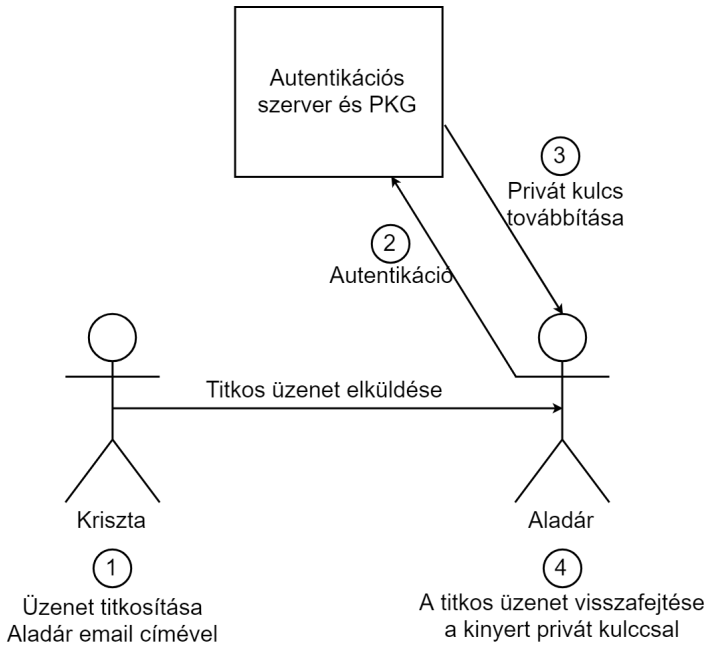
A már említett, Boneh és Franklin (2001) nevéhez fűződő rendszer volt az első, amely teljesen működőképesnek és a gyakorlatban is hatékonyan használhatónak bizonyult.

3.2.1. Boneh-Franklin Identity-based Encryption

Shamir eredeti elképzelése az volt, hogy a publikus kulcsa mindenkinek az email címe legyen. Ennek köszönhetően megspórolhatóvá válna a publikus kulcs megszerzésének költsége.

Az elgondolt séma működése egyszerű. Mikor Kriszta titkosított emailt szeretne küldeni Aladárnak, egyszerűen titkosítja azt Aladár email címével, majd elküldi. Ahhoz, hogy Aladár az üzenetet el tudja olvasni, előbb vissza kell fejtenie. Ezt úgy tudja megtenni, hogy a használt alkalmazás által specifikált módon azonosítja magát, ami után eléri a

privát kulcs generátort (PKG). A PKG felelős a felhasználók privát kulcsának elkészítéséért. Az elkészült privát kulcs eljut Aladárhoz, aki azt felhasználva vissza tudja fejteni az üzenetet. Ez a folyamat megtekinthető a 3.1. ábrán.



3.1. ábra. Az IBE működése.

A séma előbb vázolt működéséhez szükséges még egy, az ábrán nem szereplő előkészítő lépés is. Ennek részeként létrejönnek az úgynevezett publikus paraméterek, valamint a mesterkulcs. Ahhoz, hogy titkosított kommunikációt folytathassunk, először be kell szereznünk a publikus paramétereket; kiemelendő ugyanakkor, hogy erre csak egyszer van szükség, hiszen ezek a paraméterek függetlenek mind a feladótól, mind a címzettől. Míg a publikus paraméterek a rendszer összes felhasználója számára ismertek, addig a mesterkulccsal csupán a PKG rendelkezik – a privát kulcsok előállítása ugyanis csak ennek birtokában lehetséges.

Boneh és Franklin egy ennek az elképzelésnek eleget tevő rendszert dolgozott ki, amelyet négy algoritmus alkot. Ezek leírásához Martin könyvét (2008) és Kovács diplomamunkáját (2014) vettük alapul.

Az egyes algoritmusok ismertetésének és az azokat követő pszeudokódoknak a megértését elősegíti a 3.1. táblázat, amiben összefoglaltuk a különböző paraméterek jelentését.

Paraméter neve	**Típusa**	**Megjegyzés**
q	prím	-
p	prím	-
$E(\mathbb{F}_p)$	elliptikus görbe	-
G_1	az $E(\mathbb{F}_p)$ ciklikus részcsoportja	generátora P
$G_\mathbb{T}$	az $E(\mathbb{F}_p)$ ciklikus részcsoportja	generátora $e(P,P)$
e	párosítás	$e : G_1 \times G_1 \to G_\mathbb{T}$
n	egész szám	a titkosítatlan szöveg hossza
P	elliptikus görbe pontja	$P \in E(\mathbb{F}_p)$
sP	elliptikus görbe pontja	$sP \in E(\mathbb{F}_p)$
H_1	kriptográfiai hash függvény	$H_1 : \{0,1\}^* \to G_1$
H_2	kriptográfiai hash függvény	$H_2 : G_\mathbb{T} \to \{0,1\}^n$
H_3	kriptográfiai hash függvény	$H_3 : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}_q$
H_4	kriptográfiai hash függvény	$H_4 : \{0,1\}^n \to \{0,1\}^n$

3.1. táblázat. A Boneh-Franklin Identity-based Encryption paraméterei.

Setup. Ez a függvény felelős a rendszer inicializálásáért, a felhasználóhoz tartozó publikus paraméterek és a mesterkulcs előállításáért. Bemenetként a biztonsági fokot meghatározó k paramétert várja. A 3.2. táblázatban az RFC 5091 ajánlásai láthatók a különböző k értékekre vonatkozóan (Boyen & Martin, 2007). A táblázatban szereplő k értékek a hasonló biztonságot nyújtó RSA kulcshosszak méretét jelképezik, ahogy azt az NSA táblázatából is leolvashatjuk (*The Case for Elliptic Curve Cryptography*, 2009).

k **értéke**	q **bithosszúsága**	p **bithosszúsága**	**Jelölése a dolgozatban**
1024	160	512	LOWEST
2048	224	1024	LOW
3072	256	1536	MEDIUM
7680	384	3840	HIGH
15360	512	7680	HIGHEST

3.2. táblázat. Az RFC 5091 ajánlásai a biztonsági paraméternek megfelelő bithosszakra.

A *Setup* pszeudokóját az Algoritmus 2 adja meg.

Algoritmus 2 Setup

procedure SETUP(k)

q prím inicializálása ▷ k paraméternek megfelelően

p prím inicializálása ▷ k paraméternek megfelelően

$E(\mathbb{F}_p)$ elliptikus görbe inicializálása

$P = randomPoint(E(\mathbb{F}_p))$ ▷ $P \in E(\mathbb{F}_p)$

G_1 csoport generátora legyen P

$e : G_1 \times G_1 \to G_{\mathbb{T}}$ párosítás kiválasztása

$G_{\mathbb{T}}$ csoport generátora legyen $e(P, P)$

$s = randomInteger(q)$ ▷ $s \in \mathbb{Z}_q$

$H_1 : \{0,1\}^* \to G_1$ kriptográfiai hash függvény kiválasztása

$H_2 : G_{\mathbb{T}} \to \{0,1\}^n$ kriptográfiai hash függvény kiválasztása

$H_3 : \{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}_q$ kriptográfiai hash függvény kiválasztása

$H_4 : \{0,1\}^n \to \{0,1\}^n$ kriptográfiai hash függvény kiválasztása

$PublicParameters = (G_1, G_{\mathbb{T}}, e, n, P, sP, H_1, H_2, H_3, H_4)$

return $s, PublicParameters$ ▷ s a mesterkulcs

end procedure

Extract. Az azonosítóhoz tartozó privát kulcs kinyerésére szolgáló függvény. Bemenetként egy ID azonosítót vár. A privát kulcsot csak egyszer kell generálni, aztán mindaddig használható, amíg nem kompromittálódik. Ha egy privát kulcs kompromittálódik, például ellopják, az azt jelenti, hogy minden üzenet, ami az ahhoz tartozó nyilvános paraméterekkel lett titkosítva, veszélyben van. A séma nem ad támogatást ilyen veszéllyel szemben, azonban a *Setup* függvény újrafuttatásával új nyilvános paraméterek generálhatók, ekképpen az új üzenetek ismét biztonságban lehetnek.

Algoritmus 3 Extract

procedure EXTRACT(ID)

$Q_{ID} = H_1(ID)$

return sQ_{ID}

end procedure

Encrypt. Az üzenet titkosítását végző függvény. Bemenetként egy M üzenetet és egy ID azonosítót vár.

Algoritmus 4 Encrypt

procedure ENCRYPT(M, ID)
  $Q_{ID} \leftarrow H_1(ID)$
  $\sigma \leftarrow randomBits(n)$ ▷ $\sigma \in \{0,1\}^n$
  $r \leftarrow H_3(\sigma, M)$
  $C_1 \leftarrow rP$
  $C_2 \leftarrow \sigma \oplus H_2(e(rQ_{ID}, sP))$
  $C_3 \leftarrow M \oplus H_4(\sigma)$
  $C \leftarrow (C_1, C_2, C_3)$
  return C
end procedure

Decrypt. A titkos üzenet visszafejtéséért felelős függvény. Bemenetként egy C titkos üzenetet és egy sQ_{ID} privát kulcsot vár.

Algoritmus 5 Decrypt

procedure DECRYPT(C, sQ_{ID})
  $\sigma \leftarrow C_2 \oplus H_2(e(sQ_{ID}, C_1))$
  $M \leftarrow C_3 \oplus H_4(\sigma)$
  $r = H_3(\sigma, M)$
  if $C_1 \neq rP$ **then**
    return error ▷ Hibás bemenet.
  end if
  return M
end procedure

Fontos megjegyezni, hogy az *Encrypt* és *Decrypt* függvények egymás inverzét alkotják. Ez azt jelenti, hogy ha $\mathcal{M}$ jelöli az üzenetteret, akkor $\forall M \in \mathcal{M} : Decrypt(Encrypt(M, ID), sQ_{ID}) = M$ (Boneh & Franklin, 2001).

A bizonyítás a párosítás alapvető tulajdonságát használja ki, mely szerint:

$$e(rQ_{ID}, sP) = e(Q_{ID}, P)^{rs} = e(sQ_{ID}, rP).$$

4. fejezet

WebAssembly

Míg a megelőző fejezetek a dolgozat eredményét jelentő könyvtár kriptográfiai alapjait fektették le, addig ebben a fejezetben az implementációhoz használt egyik legfontosabb technológiát, a WebAssemblyt (röviden: Wasm) mutatjuk be részletesen.

Annak érdekében, hogy érezhető legyen a WebAssembly valódi jelentősége, ismertetésre kerülnek a hasonló célokat szolgáló, azonban ma már túlhaladottnak tekinthető eszközök. Ezt követően a WebAssembly áttekintését adjuk, kiemelt figyelmet szentelve azoknak az előnyös tulajdonságoknak, melyek megkülönböztetik a korábbi technológiáktól. Végül egy rövid példa zárja a fejezetet, ízelítőt adva a fejlesztési folyamatból.

4.1. Előzmények

A dinamikus weboldalak létrehozását lehetővé tevő JavaScript programozási nyelv 1995-ben jelent meg először a Netscape Navigator 2.0 böngészőben (*Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the Internet*, 1995). Bár szerveroldali alkalmazása már a kezdetektől is lehetséges volt, azonban a 2010-es évekig (a Node.js feltűnéséig) inkább a kliensoldalon játszott meghatározó szerepet, hiszen az első pillanattól kezdve támogatta a HTML oldalak manipulációját, miközben egyszerű, laikusok számára is érthető szintaxissal bírt.

Ugyanakkor nem a JavaScript az egyetlen olyan technológia, mely az elmúlt évtizedekben a dinamikus webes tartalmak elkészítését, vagy egyszerűen csak a böngészőben való, lehetőleg hatékony kódfuttatás elősegítését szolgálta. A teljesség igénye nélkül érdemes megemlékezni a következő technológiákról, melyek mindegyike ilyen, vagy olyan szempontból, de az előbbi célokra készült, így egyúttal a WebAssembly előzményének is tekinthető:

- Java Applet,

- Adobe Flash,
- ActiveX,
- Native Client (NaCl), illetve Portable Native Client (PNaCl),
- asm.js.

A következőkben e technológiák főbb jellemzőit tekintjük át: az előbbi három esetében csak érintőlegesen, míg a (P)NaCl és az asm.js esetében részletesebben.

4.1.1. Korai beépülő modulok

A Java programozási nyelvhez kapcsolódó platformfüggetlen infrastruktúrát használják ki az *appletek*, melyek más alkalmazásokba beágyazható kis programok(*JavaDocs: java.applet.Applet*, 2018). Az egyik legnépszerűbb beágyazó környezetet a webböngészők jelentik, melyek beépülő modulok segítségével képesek az *appletek* megjelenítésére (*Java Plug-In Technology*, 2018). Természetesen az *appletek* forráskódja a beágyazó böngészőtől teljesen független. Kiemelendő, hogy az *appletek* alapértelmezés szerint egy zárt, úgynevezett *sandbox* környezetben futnak, mely megakadályozza, hogy kártékony tevékenységeket hajtsanak végre a felhasználó tudta nélkül (*The Java Tutorials: Java Applets*, 2018).

A Flash először csak gyors rajzolási és animációs képességekkel kívánta felruházni a böngészőket, az évek során azonban egy átfogó multimédia platformmá fejlődött (Gay, 2001). Létrejöttében kulcsszerepet játszott az *appletek* multimédiás célokra történő alkalmatlansága. A Flash animációk böngészőben történő megjelenítéséhez szintén egy beépülő modul szükséges, ez a Flash Player (*Adobe Flash Player*, 2018).

Említésre méltó a Microsofthoz köthető ActiveX Technologies keretrendszer, mely úgynevezett ActiveX Controlok útján tette lehetővé dinamikus komponensek és objektumok beágyazását a HTML oldalakba (*Microsoft Announces ActiveX Technologies*, 1996). Habár az ActiveX Controlok is egy beépülő böngésző modult igényelnek a megfelelő működéshez, azonban a technológia jelentősen eltér az előzőleg látottaktól. Egyfelől az egyes komponensek platformfüggő binárisok formájában kerülnek terjesztésre (Grimes, 2001), másfelől azok közvetlenül kerülnek lefuttatásra, így tetszőleges műveleteket végrehajthatnak a felhasználó számítógépén (*Designing Secure ActiveX Controls*, 2017).

Ma már ezen három technológia mindegyike túlhaladottnak tekinthető. Az Applet API a Java 9-ben már *deprecated* annotációval bír (*JEP-289*, 2017), a Flash támogatását az Adobe 2020-ban végleg beszünteti (*Flash & The Future of Interactive Content*, 2017), az ActiveX pedig az úgynevezett *evergreen* böngészők egyikében sem támogatott már, még a Microsoft Edge-ben sem (*A break from the past*, 2015).

4.1.2. Native Client, Portable Native Client

Míg az előzőleg felsorolt technológiák inkább a böngésző, mint alkalmazásfejlesztési platform hiányosságait kívánták pótolni, addig a Google ernyője alatt megszülető Native Client (röviden NaCl) a teljesítményre fókuszál. A NaCl a natív (azaz például C vagy C++ nyelven írt) programok hatékony futtatását tette lehetővé a böngésző által biztosított *sandbox* környezetben (Yee és mtsai., 2009). A NaCl platformfüggetlen testvére a Portable Native Client (röviden PNaCl), mely a HTML oldalakba való beágyazást is biztosítja (*NaCl and PNaCl*, 2017).

A PNaCl modulokat (*pexe*) egy LLVM-alapú fordító segítségével lehet előállítani. Érdekesség, hogy az így elkészült modul a fordító belső, átmeneti reprezentációját használja, a tartalmazott kód ugyanis LLVM IR (*intermediate representation*) nyelvű. A létrejött modulok végrehajtása két lépésből áll. Először a böngészőbe épített *ahead-of-time* (AOT) fordító platformfüggő kódot készít, majd pedig a NaCl modulok futtatásához is használt *sandbox* ténylegesen lefuttatja a kódot (*Native Client: Technical Overview*, 2017).

Noha a PNaCl valóban alkalmas volt a böngészőn belüli gyors és biztonságos végrehajtásra, a Google Chrome-on kívül sosem terjedt el igazán. Ma már a Google is inkább a WebAssembly használatát javasolja (Nelson, 2017).

4.1.3. asm.js

Az egyetlen programozási nyelv, mely stabilan elérhető az összes platform összes böngészőjében, a JavaScript. Ezen felismerésre alapozva kialaultak olyan eszközök, melyek valamilyen másik programozási nyelv kódbázisát fordítják a böngészőben futtatható JavaScriptre (*List of languages that compile to JS*, 2018). Ezek közül is kiemelkedik az asm.js, mely a böngészőben történő nagyteljesítményű kódvégrehajtást helyezi előtérbe.

A futási sebesség növekedése két tényező eredménye. Egyfelől a fordítás első lépése az LLVM infrastruktúrán keresztül valósul meg, mely eleve egy optimalizált átmeneti reprezentációt állít elő. Ezt az Emscripten a JavaScript egy rendkívül szűk részhalmazára fordítja, mely a hatékonyság mögötti második faktor. Ez a részhalmaz ugyanis a JavaScript dinamikus jellemzőit elhagyja, így a fokozatosan optimalizáló *just-in-time* (JIT) fordító helyett a kód fordításához rögtön egy, a böngésző teljes optimalizálási eszközkészletét kihasználó AOT fordító használható (Zakai, 2015). Kiemelendő azonban, hogy az asm.js alkalmazása akkor is sebességnövekedéssel jár, ha a böngésző csak JIT-et használ.

Habár az asm.js nem tekinthető elavultnak, a WebAssembly elterjedésével párhuzamosan jelentősége várhatóan csökkenni fog. Az Emscripten például alapértelmezés szerint már WebAssembly kódot állít elő asm.js helyett (*Emit WebAssembly by default instead of asm.js*, 2018).

4.2. A WebAssembly, mint modern célplatform

A megelőző pontok azon technológiák közül szemezgettek, melyek a '90-es évek közepétől kezdődően a web, mint alkalmazásfejlesztési platform gazdagítását, felgyorsítását szolgálták. Szerepük a böngészők (és a web) fejlődésében vitathatatlan, hiszen a ma ismert szabványos és modern Web API-k funkcionalitását nyújtották, jóval azok megjelenése előtt. Ugyanakkor a megfelelő W3C szabványok feltűnésével és elterjedésével ezek a bővítmények szükségtelenné, mi több, a fejlődés akadályozóivá váltak.

A multimédiás lehetőségeket biztosító bővítmények (például: Flash, Shockwave, Silverlight, ActiveX) visszaszorulásában többek között a HTML5, a CSS3, az SVG, és a WebGL játszott fontos szerepet. A hatékony, natívhoz mérhető sebességű kódvégrehajtásra pedig a WebAssembly nyújt az egyes platformokon és böngészőkön átívelő modern, egységes megoldást.

4.2.1. A WebAssembly specifikáció

A WebAssembly a W3C WebAssembly Community Group által 2015-től fejlesztett virtuális ISA (*instruction set architecture*), melynek elsődleges célja a weben történő nagysebességű kódvégrehajtás, ugyanakkor tetszőleges környezetbe beágyazható (*WebAssembly Specification*, 2018). Szemben a korábbi technológiákkal, melyeket rendre egyetlen gyártó tartott befolyása alatt (beleértve a specifikáció fejlesztését, esetleg a licencelést), a WebAssembly egy széleskörű iparági összefogással létrehozott nyílt W3C szabványon alapul. Támogatottságát jól jellemzi, hogy rendkívül fiatal volta ellenére már a webet látogató felhasználók több mint 75%-a rendelkezik a WebAssemblyt támogató böngészővel (*Can I use WebAssembly?*, 2018). A WebAssembly tehát közvetlenül a web, mint platform részévé vált, ami éles kontrasztban áll a beépülő modulok és bővítmények világával.

A specifikáció tervezése során kiemelt hangsúly került a megfelelő szemantika és reprezentáció kialakítására, tanulva az előzménynek tekinthető eszközök előnyös vagy éppen kellemetlen tulajdonságaiból. A következőkben a specifikációt (*WebAssembly Specification*, 2018) alapul véve ismertetjük ezeket a célokat. A technológia rövidebb, de egyúttal tágabb kontextusba helyezett áttekintését adja Haas és munkatársai témába vágó írása (2017).

Szemantika

A szemantika kialakítását a következő tervezési célok hajtották:

Gyors. A WebAssembly modulok megközelítőleg a natív programokra jellemző sebességgel kerülnek végrehajtásra.

Biztonságos. A modulok egy *sandbox* környezetben futnak, mely garantálja a biztonságos végrehajtást és az izolációt, kiemelt figyelmet szentelve a memóriakezelésnek. A WebAssembly programok által használt memória zárt és elkülönített, azaz nem férhetnek hozzá a rendszer vagy a többi folyamat memóriaterületéhez. Ezen felül a modulok kódja a futtatás előtt validálásra kerül, biztosítandó azok szabályos voltát.

Jól definiált. A szemantika pontosan leírja a programok működését és viselkedését, lehetővé téve ezzel akár a végrehajtás formális elemzését is. A platform javarészt determinisztikus, lokális nemdeterminizmus mindössze hat jól definiált esetben fordulhat elő.

Hordozható. A hordozhatóság két irányból is megközelíthető. Egyfelől a WebAssembly hardver- és platformfüggetlen, hasonlóan a JVM-hez. Bármely platform, amihez rendelkezésre áll egy WebAssembly-környezet, képes a Wasm kódok futtatására. Másik irányból közelítve, a WebAssembly forrásnyelv-független. A szemantika teljesen agnosztikus, nem részesít előnyben semmilyen programozási paradigmát vagy objektummodellt, így tetszőleges programozási nyelvről azonos feltételek mellett fordíthatunk Wasm kódot.

Nyílt. Biztosított a modulok és tetszőleges beágyazó környezet közötti interoperabilitás. A WebAssembly egyetlen környezetet sem helyez előtérbe, így a JavaScript API (Daniel Ehrenberg, 2018a) és a Web API (Daniel Ehrenberg, 2018b) is a szabványhoz kapcsolódó külön dokumentumban kapott helyet.

Reprezentáció

A böngésző, mint elsődleges beágyazó környezet hatása a szemantikán is tetten érhető, még nyilvánvalóbb azonban ez a hatás a reprezentáció legfontosabb céljait tekintve:

Tömör. A szabvány az emberek által is olvasható (és írható) szöveges reprezentáció mellett meghatároz egy tömör bináris formátumot is, mely helytakarékosabb, mint a minifikált JavaScript kód, sőt, a natív binárisoknál is kevesebb tárhelyet igényelhet. A motiváció emögött egyértelmű: minél kevesebb adatot kelljen a hálózaton keresztül továbbítani. A bináris reprezentáció tömörsége köszönhető többek között annak, hogy a Wasm verem-alapú virtuális gépet használ. Az ilyen típusú virtuális gépekhez készült kód általában kevesebb tárhelyet igényel, mint például a regiszter-alapú virtuális gépek kódja (Friedman & Wand, 2008).

Moduláris. A WebAssembly kód modulok formájában tehető közzé. Ezek a modulok szabadon exportálhatják tartalmukat, valamint importálhatják más modulok deklarációit. Ily módon a kódbázisok szétbonthatók kisebb darabokra, melyek közül

a ritkán változó modulok akár kliensoldalon cache-elhetők, ezzel is csökkentve a hálózati adatforgalmat.

Streamelhető és párhuzamosítható. A modulok dekódolása, validálása és fordítása a teljes kód átvitele előtt megkezdődhet. A modulok különböző részeinek feldolgozása akár egymástól függetlenül, párhuzamosítva is történhet.

4.2.2. Teljesítmény

A WebAssembly egyik legfontosabb célkitűzése a natív programokét megközelítő teljesítmény elérése. Érdemes a teljesítményt befolyásoló tényezők elemzését a JavaScripttel, illetve az asm.js-szel történő összehasonlítás formájában megtenni. Azaz, hogyan képes a WebAssembly még ezeknél is nagyobb teljesítményt kínálni?

A válasz első összetevője lehet az úgynevezett *startup time*. Ez tartalmazza a futtatandó kód letöltését, feldolgozását és fordítását, egészen az első lefuttatott utasításig (Zakai, 2017). A Wasm bináris reprezentációja rendkívül helytakarékos, azaz gyorsabban letölthető. Ezt követően a feldolgozása és elemzése (*parsing*) is kevesebb időt emészt fel, hiszen eleve gépi olvasásra szánt formátumban van, szemben a JavaScripttel (Clark, 2017b).

Mivel a WebAssembly kód platformfüggetlen, a futtatást megelőzően az adott beágyazó platformnak megfelelő bináris kódot kell fordítani belőle. Ez megtehető egy JIT vagy egy AOT fordító segítségével. A JIT-et használva a *startup time* alacsonyan tartható, a kevésbé hatékony optimalizáció árán. A WebAssembly *startup time*-ja ebben az esetben tehát biztosan alacsonyabb, mint a JavaScripté, még az asm.js-t tekintve is (Zakai, 2017).

Megfontolandó lehet a JIT helyett az AOT fordítást választani, hiszen a WebAssembly kódból már az első pillanattól kezdve optimális natív kód fordítható. Míg a JavaScript esetén azért van szükség a JIT fordításra, hogy a végrehajtó környezet a futási információkat használva kiválaszthassa az alkalmazható optimalizációkat (Clark, 2017a), addig a WebAssembly statikus természete miatt erre nincsen szükség, rögtön alkalmazható az AOT fordítás. Bár ez az asm.js esetén is lehetséges, azonban a WebAssembly általában jobban optimalizálható (Zakai, 2017).

Tekintve, hogy a WebAssembly egy, a natív kódhoz közel álló bájtkódot határoz meg, hatékonyabb natív kódra fordítható, mint a JavaScript. Ez azt jelenti, hogy a WebAssembly jobban ki tudja használni a hardverspecifikus sajátosságokat, a rendelkezésre álló processzor utasításkészletét (Zakai, 2017).

A felsoroltaknak köszönhetően a WebAssembly képes még az asm.js-nél is gyorsabb végrehajtást kínálni, megközelítve ezzel a natív programok végrehajtási sebességét, ami megnyitja a lehetőséget a számításigényes programok webes megjelenése előtt.

4.2.3. Támogatott programozási nyelvek

A szemantikát befolyásoló tervezési célok között szerepelt a nyelvfüggetlenség igénye. Ez azt jelenti, hogy tetszőleges programozási nyelvből kiindulva létrehozható WebAssembly kód, függetlenül az adott nyelv sajátosságaitól.

Habár számos nyelvhez megkezdték a szükséges eszközkészlet fejlesztését, igazán stabil támogatásra azonban csak a következő nyelvek esetén számíthatunk:

C/C++. A C, illetve C++ nyelven írt kódbázisok futtatásának támogatása már a WebAssembly MVP (*Minimum Viable Product*) specifikációjában is szerepelt. Ennek megfelelően e két nyelv támogatása tekinthető a legstabilabbnak. A modulok előállítása történhet például a Cheerp[1], vagy az asm.js-hez is használt Emscripten[2] segítségével.

Rust. A Rust[3] egy rendszerközeli prgramozási nyelv (*systems programming language*), mely a sebesség mellett a biztonságos szál- és memóriakezelést helyezi előtérbe. A Rust nyelven írt programok WebAssemblyre történő fordítására alkalmas például a wasm-pack[4].

Az elkövetkező évek során feltehetően további programozási nyelvek is jobb WebAssembly támogatással fognak bírni, jelenleg azonban még csak a fentiek tekinthetők megbízható, akár *production* környezetben is alkalmazható megoldásnak.

4.3. Demonstráció

A WebAssembly mögött álló gondolatok megismerése után nézzük meg a technológia működését a gyakorlatban is! Ennek demonstrálása a következőkben egy rendkívül egyszerű C nyelvű könyvtár fordításán, majd pedig webes futtatásán keresztül fog megtörténni. Komplexebb projektek esetén az itt leírtak kiegészülhetnek további lépésekkel és beállításokkal, így a bemutatott példa inkább csak ízelítőül szolgál. Az érdeklődő Olvasó számára javasolt a GitHubon található Awesome Wasm[5] tároló felkeresése, mely a WebAssemblyt használó, vagy azzal kapcsolatos projektek folyamatosan bővülő gyűjteménye.

[1] `https://leaningtech.com/cheerp`
[2] `http://emscripten.org`
[3] `https://www.rust-lang.org`
[4] `https://rustwasm.github.io/wasm-pack`
[5] `https://github.com/mbasso/awesome-wasm`

4.3.1. A C nyelvű könyvtár

A könyvtár, melyből WebAssembly modult szeretnénk létrehozni, mindössze egyetlen forrásfájlból áll (`library.c`), melynek tartalma a 4.1. kódrészletben olvasható.

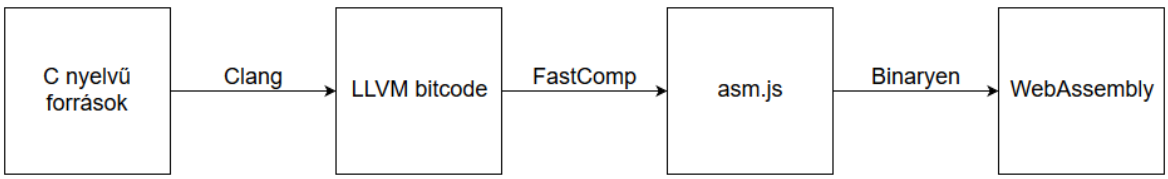
```
1  int factorial(const int n)
2  {
3      int result = 1;
4
5      for (int i = 2; i <= n; ++i)
6      {
7          result *= i;
8      }
9
10     return result;
11 }
```

4.1. Kódrészlet. A fordítandó `library.c` fájl.

Az egyetlen függvény, melyet a könyvtár biztosít, a faktoriálist számító `factorial`. Vegyük észre, hogy ez egy teljesen átlagos, C nyelven írt függvény, nem tartalmaz semmilyen, a WebAssemblyhez kapcsolódó utasítást, definíciót vagy pragmát!

4.3.2. Fordítás

A fordításhoz a C/C++ támogatásnál említett Emscripten eszközkészletet fogjuk használni. A fordítás fázisait a 4.1. ábrán láthatjuk.



4.1. ábra. C → Wasm fordítás Emscriptennel.

A folyamat első lépéseként a Clang fordító az LLVM belső reprezentációjára hozza az eredeti C nyelvű forrásokat. Ez egy könnyen optimalizálható formátum, melyből a FastComp asm.js kimenetet állít elő. Habár ez a köztes lépés nem lenne feltétlenül szükséges, azonban a köztes reprezentációból WebAssemblyt készítő LLVM *backend* még instabilnak tekinthető, így egyelőre érdemesebb ezt az utat választani. A folyamat utolsó lépése a tényleges WebAssembly modul elkészítése, melyről a Binaryen gondoskodik. Természetesen a köztes lépések a felhasználó elől rejtve zajlanak.

Visszatérve a példához, a `library.c` fordításához szükséges parancsot a 4.2. kódrészlet tartalmazza. Az `EXPORTED_FUNCTIONS` kapcsolóval külön meg kell adnunk a modul kliensei által is látható függvények listáját.

```
emcc library.c -o library.js \
     -s EXPORTED_FUNCTIONS='["_factorial"]' \
     -s EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]'
```

4.2. Kódrészlet. A `library.c` fordítása Emscriptennel.

A futás eredményeként két fájl áll elő: a `library.wasm` és a `library.js`. Előbbi a C könyvtárból készült Wasm modul, míg utóbbi egy segédeszközöket tartalmazó JavaScript fájl, mely megkönnyíti a modul JavaScriptből történő felhasználását.

4.3.3. Beágyazás HTML-be

Az elkészült modul HTML kódba történő beágyazását a 4.3. kódrészlet szemlélteti. Ezt egy HTML fájlként elmentve, majd megfelelő (azaz WebAssemblyt támogató) böngészőben megnyitva a „*The factorial of 3 is 6.*” szöveget fogjuk látni.

```
1  <!doctype html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Factorial</title>
6  </head>
7  <body>
8      <p id="result"></p>
9      <script type="text/javascript">
10         var Module = {
11             onRuntimeInitialized() {
12                 const factorial = Module.cwrap('factorial', 'number', ['number'])
13
14                 document.getElementById('result').textContent =
15                     `The factorial of 3 is ${factorial(3)}.`
16             }
17         }
18     </script>
19     <script async src="library.js"></script>
20 </body>
21 </html>
```

4.3. Kódrészlet. A beágyazó HTML kód.

Ugyan a modul beágyazása a 19. sorban található `script` elemben történik, ez még csupán a WebAssembly kód feldolgozásáról, validálásáról és JIT vagy AOT fordításáról gondoskodik.

A `factorial` függvény tényleges futtatásának módját a 12-15. sorokban láthatjuk. Először a 12. sorban a `cwrap` függvény segítségével egy JavaScript függvényt készítünk az exportált `factorial` függvényből. A `cwrap` az exportált függvény neve mellett annak visszatérési típusát, valamint paramétereinek típusát várja. Ezt követően a visszaadott függvény már a szokásos JavaScript függvényekkel azonos módon hívható, amit a 15. sorban láthatunk.

Érdemes még a beágyazó kódban található `Module` objektumot is megemlíteni (10. sor). Alapesetben az Emscripten által kibocsátott segédkönyvtár egy `Module` nevű objektumot exportál. Ugyanakkor, ha a beágyazó kód már tartalmaz egy ilyen nevű változót, akkor a segédkönyvtár ezt fogja felhasználni. Az ezen az objektumon definiált `onRuntimeInitialized` függvény akkor kerül meghívásra, amikor a WebAssembly modul elérhetővé vált, azaz a futtató környezet felkészült a kódvégrehajtásra.

5. fejezet

CryptID

Az előző fejezetek előbb a kriptográfiai, majd a technológiai alapokat vezették be. Ebben a fejezetben a dolgozat eredményét jelentő programkönyvtár – a CryptID – kerül részletesen ismertetésre.

A fejezet felépítése a következő: Először magas szinten vázoljuk, hogy mi is pontosan a CryptID, miben jelent újdonságot, milyen megfontolások állnak mögötte. Ezt követi a könyvtár struktúrájának alapos bemutatása, rétegről rétegre haladva. Az utolsó előtti szekció egyfajta útmutatóként szolgál a CryptID felhasználásához, integrálásához. Végül a fejezetet a könyvtár teljesítményének elemzése zárja.

5.1. Mi a CryptID?

A CryptID egy nyílt forrású IBE megoldás, mely az RFC 5091-ben meghatározott Boneh-Franklin sémát veszi alapul. Újszerűsége két irányból is megközelíthető, egyrészt az implementációt, másrészt az IBE sémát tekintve.

A megvalósításban rejlő újdonság, hogy a CryptID WebAssembly alapokon működik, aminek köszönhetően nemcsak a szerveroldalon, hanem a kliensoldalon, vagy akár a webtől teljesen elszakadva nyújt platformfüggetlen és hatékony titkosítási megoldást.

Az IBE sémához hozzátett újítás a publikus kulcsban keresendő. A CryptID strukturált publikus kulcsokra épül, melyekben az egyedi azonosítón felül tetszőleges metaadat elhelyezhető.

A következőkben az előbb felsorolt sajátosságokat fejtjük ki részletesen, mindenütt kitérve a háttérben álló motivációkra is.

5.1.1. Platformfüggetlen működés

A CryptID formájában egy olyan könyvtárat szerettünk volna létrehozni, mely hatékony kliensoldali titkosítást tesz lehetővé. Mindezt olyan formában szerettük volna megvalósítani, hogy ugyanaz a megoldás alkalmazható legyen asztali gépeken, tableteken és mobiltelefonokon egyaránt. Ezt az erőfeszítést az a motiváció hajtotta, hogy nem tudunk olyan nyílt forrású IBE könyvtárról, mely módosítás nélkül (*out-of-the-box*) használható lenne ezen platformok mindegyikén.

Adott volt tehát a célkitűzés: egy böngészőben futtatható, hatékony programkönyvtár létrehozása. Korábban a JavaScript volt az egyetlen olyan technológia, mely ilyen mértékű platformfüggetlenséget kínált. Azonban az egyes böngészőkben található JavaScript motorok jelentősen eltérő optimalizációkat alkalmazhatnak, így ami az egyik böngészőben gyorsan fut, az elképzelhető, hogy egy másikban jóval gyengébb teljesítményt nyújt. Erre a problémára ugyan megoldást kínál az asm.js, mely egyszerűségénél fogva könnyebben és egyértelműbben optimalizálható, azonban ez korántsem tekinthető szilárd és jól támogatott szabványnak.

Rátaláltunk azonban az előző fejezetben ismertetett WebAssembly szabványra, mely pontosan azt nyújtotta, amire szükségünk volt: hordozható binárist és gyors végrehajtást. A WebAssemblynek köszönhetően kihasználhattuk azt is, hogy C-ben számos régóta fejlesztett és alaposan tesztelt matematikai, illetve kriptográfiai programkönyvtár áll rendelkezésre. Ilyen a GMP (Granlund & the GMP development team, 2016) és az OpenSSL (*OpenSSL Cryptography and SSL/TLS Toolkit*, 2018) is, melyek a CryptID alapját képzik.

5.1.2. Strukturált publikus kulcs

Az IBE lényege, hogy a publikus kulcs egy adott domainen belül valamilyen entitást egyértelműen azonosít. Legegyszerűbb példa erre egy email cím, vagy adott rendszeren belüli felhasználónév. Boneh és Franklin azonban a róluk elnevezett sémát leíró cikkükben (2001) említenek olyan alkalmazásokat is, melyek a publikus kulcsot további metaadatokkal egészítik ki. Ilyen metaadat lehet például az aktuális év, mely egyfajta érvényességet rendel a publikus kulcshoz és így áttételesen a hozzátartozó privát kulcshoz is. Az említett cikkben ez a metaadat egyszerűen az azonosítóhoz konkatenálva jelenik meg a publikus kulcsban: „`bob@company.com` ‖ `current-year`”.

A metaadatok beágyazásának ötletét nagyszerűnek találtuk, azonban úgy éreztük, hogy a konkatenáció szükségtelen kötöttséget erőltet a publikus kulcsra: az egyes mezőknek mindig azonos sorrendben kell szerepelniük. Természetesen ezen közvetlenül nem változtathatunk, hiszen a titkosítás, majd a visszafejtés csak akkor működik az elvártnak megfelelően, ha a publikus kulcs mindig bitpontosan azonos.

E megszorítást csak az absztrakció szintjének megemelésével kerülhettük meg. A CryptID ennek folytán JavaScript objektumokat használ publikus kulcsként. Az objektumok publikus kulcsra történő leképezése a könyvtár implementációs részletei közé tartozik, ezzel a CryptID-et integráló fejlesztőnek nem kell foglalkoznia. Ily módon a CryptID megszünteti a sorrendi kötöttség okozta terhet, mi több, a fejlesztőknek azzal sem kell törődniük, hogy a publikus kulcsot reprezentáló objektumból hogyan lesz bitsorozat – a konverzió a kulisszák mögött történik.

5.1.3. Nyílt forrású, RFC-alapú implementáció

Egy adott kriptorendszerhez megbízható implementációt készíteni rendkívül nehéz feladat. Hiába áll ugyanis rendelkezésünkre egy matematikailag bizonyítottan biztonságos kriptorendszer, annak megvalósítása során könnyen véthetünk olyan hibákat, melyek sebezhetőségeket nyitnak. Ezek a sebezhetőségek fakadhatnak például programozási hanyagságból (például a bemenet nem megfelelő ellenőrzése), vagy matematikai figyelmetlenségből, tudatlanságból (támadható elliptikus görbe használata).

Számos gyakori hiba megelőzhető azonban, ha valamilyen nyílt szabvány vagy ajánlás alapján készítjük el az implementációnkat. A CryptID esetében is így tettünk: az RFC 5091-et használtuk fel, mely a Boneh-Franklin és a Boneh-Boyen IBE rendszerek lehetséges implementációját írja le egy bizonyos szuperszinguláris elliptikus görbe felett (Boyen & Martin, 2007). Ezek közül a CryptID a Boneh-Franklin rendszerre épül.

Az említett RFC részletes pszeudokódot biztosít az IBE-t felépítő főbb algoritmusokhoz (*Setup, Extract, Encrypt, Decrypt*, és alacsonyabb szintű társaik), ajánlásokat ad öt biztonsági szintre vonatkozóan (vö. 3.2. táblázat), lehetővé téve ezzel a rendszer parametrizálását, valamint tesztadatokat tartalmaz, melyek elősegítik az implementációk tesztelését. Ezek a tesztadatok hatalmas segítséget jelentettek a fejlesztés során, hiszen a teljes rendszer lekódolása előtt meg tudtunk bizonyosodni az azt felépítő kisebb rutinok helyes működéséről is.

A CryptID nemcsak az ajánlásnak megfelelő, hanem egyúttal nyílt forrású is. Természetesen a nyílt forrású szoftverek nem lesznek automatikusan biztonságosabbak, mint zárt forrású társaik, azonban a transzparenciának köszönhetően a kód könnyebben és többek által átvizsgálható és ellenőrizhető, így összességében a nyílt forrás hozzájárulhat a megbízhatóbb implementáció létrejöttéhez (Wheeler, 2015).

5.2. A CryptID felépítése

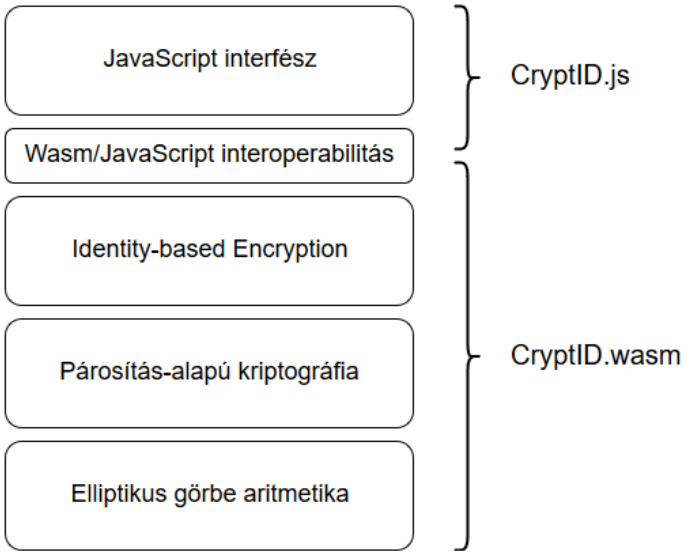
Magas szintről szemlélve, a CryptID három komponensből áll:

CryptID.ref. Java nyelven írt referencia-implementáció. Az egyetemi tanulmányaink során leggyakrabban a Java nyelvvel volt alkalmunk dolgozni, ennek folytán Javaban rendelkezünk a legmagabiztosabb tudással. Ezt kihasználva, először elkészítettük az IBE Java nyelvű megvalósítását, melyet aztán felhasználhattunk a WebAssembly implementáció helyességének ellenőrzéséhez.

CryptID.wasm. Az IBE-t alkotó rutinokat tartalmazó WebAssembly modul. A forrásnyelv C, ezt fordítjuk át Emscripten segítségével WebAssemblyre (ld. 4.3.2.).

CryptID.js. A WebAssembly modult becsomagoló JavaScript könyvtár, mely egy könnyen használható interfészen keresztül teszi elérhetővé az IBE rutinokat.

E három komponens valójában két különálló, bár szemantikát tekintve azonos IBE-megoldást alkot. Az egyiket a CryptID.ref önmagában, míg a másikat a CryptID.wasm és arra építve a CryptID.js. A továbbiakban a CryptID.ref ismertetését elhagyjuk, és a CryptID néven a CryptID.wasm és a CryptID.js komponensek együtteséből formált könyvtárra fogunk hivatkozni. E könyvtár struktúráját az 5.1. ábra szemlélteti.



5.1. ábra. A CryptID felépítése.

A következőkben lentről felfelé haladva részletesen bemutatjuk az egyes rétegeket.

5.2.1. Elliptikus görbe aritmetika

Az IBE egy elliptikus görbe kriptográfián alapuló titkosítási rendszer, ezért az elliptikus görbe aritmetika képzi a CryptID magját is. A 2. fejezetben összefoglaltuk az elliptikus görbék legfontosabb matematikai tulajdonságait, dolgozatunk ezen részében pedig az implementációhoz szükséges ismereteket tárgyaljuk.

Az elliptikus görbék matematikai háttere hosszú múltra tekint vissza, és számos elliptikus görbe aritmetikát megvalósító könyvtár áll rendelkezésre. Természetesen merül fel a kérdés, hogy ezek ellenére miért készítettünk saját implementációt?

Úgy találtuk, hogy az általunk vizsgált könyvtárak mindegyike rendelkezett olyan negatív tulajdonsággal (jelentős méret, átláthatatlan implementáció, hiányos dokumentáció), ami miatt úgy döntöttünk, hogy egy saját, pehelysúlyú réteget hozunk létre, ami megfelel minden elvárásunknak.

Elliptikus görbe aritmetika tulajdonságai

Az elliptikus görbe aritmetikát megvalósító rétegnek a legfontosabb tulajdonsága, hogy az RFC 5091-ben ajánlott *Type-1* elliptikus görbékre van optimalizálva. A *Type-1* osztály olyan görbéket takar, melyek alakja $E(\mathbb{F}_p) : y^2 = x^3 + 1$, ahol $p \equiv 11 (\mathrm{mod}\ 12)$ tetszőleges prím. Ezek a görbék a szuperszinguláris görbék egy részcsoportját képezik.

Az $\mathbb{F}_p$-beli elemek ábrázolására a GMP aritmetikai könyvtár véges test támogatását használtuk fel, ami egy rendkívül kiforrott és jól tesztelt megoldást nyújt.

A réteg képes $\mathbb{F}_{p^2}$-beli elemek ábrázolására is, amelyek lényegében $\mathbb{F}_p$-beli elemek rendezett párjai. Egy ilyen rendezett párt jelölünk (a_0, a_1)-el, amit értelmezhetünk az $a_0 + a_1 \cdot i$ komplex számnak, ahol $i^2 = -1$.

Ezzel az értelmezéssel egyszerűen a komplex aritmetikát alapul véve végezhetünk műveleteket $\mathbb{F}_{p^2}$ felett, annyi különbséggel, hogy minden esetben modulóját kell venni az eredményeknek, hogy a műveletek zártak maradjanak. Az ilyen műveletek kivitelezését a GMP alapú $\mathbb{F}_p$ test feletti aritmetikára építve végeztük el.

Implementálásra került $\mathbb{F}_{p^2}$ feletti elemek egymással és skalárral vett összeadása, additív inverz képzése, elemek egymással és skalárral való szorzása, hatványozása egész számmal és multiplikatív inverz képzése.

Az említett $\mathbb{F}_{p^2}$ aritmetika implementálására azért volt szükség, mert a kriptográfiában az egyszerű Tate párosítás gyakran nem alkalmazható. Ugyanakkor, ha módosítjuk annyiban, hogy az egyik bemenete egy torzítási leképezése az elliptikus görbe egy pontjának, akkor biztosan két egymástól lineárisan független ponttal dolgozhatunk, kizárva az elfajultságot.

Elliptikus görbe pontjainak ábrázolása

A réteg két lehetőséget nyújt a görbe pontjainak ábrázolására. Egyik az affin térben való ábrázolás, a másik lehetőség pedig a projektív tér használata. Egy alap IBE implementációhoz elegendő lenne az affin pontábrázolás is, azonban a projektív koordináták használata egyszerű optimalizációs lehetőségeket nyújt.

Fontos megjegyezni, hogy több módja is van a projektív pontábrázolásnak, melyek közül mi az RFC 5091 ajánlását követve a Jacobi projektív módszert implementáltuk. Amíg az affin ábrázolás esetében a pontunkat egy (x, y) számpáros alkotja, a Jacobi projektív módszer egy (x, y, z) számhármast használ.

Miben is rejlik az utóbbi módszer hatékonysága? Nos, ha M, S és I rendre a szorzás, négyzetre emelés és invertálás műveleteit jelölik, akkor az 5.1. táblázatban látható, hogy az egyes módszerek és görbe műveletek esetén melyik test-műveletet hányszor kell végrehajtani.

A lényegi különbséget az invertálás elhagyása jelenti a görbe műveletek esetén, mert a szorzás és invertálás számításigényének aránya általánosan $80 : 1$ (Nyakacska, 2016), míg a szorzás és négyzetre emelés esetén $10 : 8$ (Bernstein & Lange, 2008).

Pontábrázolás típusa	**Összeadás**	**Duplázás**
affin	$I + 2M + 2S$	$I + 2M + 1S$
Jacobi projektív	$12M + 4S$	$4M + 6S$

5.1. táblázat. Elliptikus görbén végzett műveletek költsége (Martin, 2008).

A tárgyalt ábrázolási módok mindegyikével a következő elliptikus görbe műveletek végezhetők:

Pont duplázás. Egyszerű pont duplázást megvalósító algoritmus, melyet a Martin könyvében leírtak alapján implementáltunk.

Pont összeadás. Az implementált algoritmus forrása megegyezik a duplázás esetében használttal.

Pont szorzása skalárral. A *Double-and-Add* algoritmust megvalósító skaláris szorzás került implementálásra a rétegben, ami megfelelőnek mondható, de még optimalizálható sebességet nyújt.

Pontábrázolások közötti konverzió. A réteg részét képezi a két említett pontábrázolás közötti konverzió, ami affinból Jacobi projektívbe való átalakítás esetén $(x, y) \rightarrow (x, y, 1)$, fordított irányban pedig $(x, y, z) \rightarrow (x/z^2, y/z^3)$ módosítást végez.

5.2.2. Párosítás-alapú kriptográfia

A BF-IBE-nek és így a CryptID-nek is kihagyhatatlan része a párosítás műveletének megvalósítása. Az RFC 5091 ajánlását követve a réteg a Tate párosítást tartalmazza.

Az implementált Tate párosítás két $E(\mathbb{F}_p)$-beli pontot képez le egy $\mathbb{F}_{p^2}$-beli elemre. Az implementációt az RFC 5091 (2007), Martin könyve (2008) és Lynn doktori disszertációja (2007) alapján készítettük el.

5.2.3. Identity-based Encryption

A CryptID fő funkcionalitását megvalósító réteg. Tartalmazza a BF-IBE négy fő függvényét (*Setup*, *Extract*, *Encrypt*, *Decrypt*), amelyeket az Identity-based Encryption című fejezetben ismertettünk.

Az implementáció során az RFC 5091 ajánlásait követtük, amelyek nem csupán az említett függvénynégyesre vonatkoznak, hanem az azok működtetéséhez szükséges algoritmusokra is.

Ilyenek az Identity-based Encryption című fejezet pszeudokódjaiban megjelent hash függvények:

H1 - *hashToPoint* $\{0,1\}^* \to G_1$

H2 - *canonical* $G_{\mathbb{T}} \to \{0,1\}^n$

H3 - *hashToRange* $\{0,1\}^n \times \{0,1\}^n \to \mathbb{Z}_q$

H4 - *hashBytes* $\{0,1\}^n \to \{0,1\}^n$

A felsorolt hash függvények alapját a *canonical* kivételével a Secure Hash Algorithms (SHA) hash függvény alkotja, amelyet számunkra az OpenSSL könyvtár biztosított. Az előbbi könyvtárat nem csupán az SHA implementációja miatt használjuk, hanem a kriptográfiailag biztonságos és jól tesztelt random szám generátora is nagy segítséget nyújtott számunkra.

5.2.4. Wasm/JavaScript interoperabilitás

A megelőző három réteg együttesen egy C nyelvű IBE implementációt alkot, mely akár WebAssemblyre történő fordítás nélkül is felhasználható, például natív alkalmazások készítéséhez. Természetesen a CryptID-ben nem mint natív könyvtár, hanem mint WebAssembly modul kerülnek elhelyezésre ezek a rétegek, egy JavaScript interfész mögé rejtve.

Felmerülhet a kérdés, hogy miért szükséges egy ilyen absztrakció, ha egyszer a WebAssemblyt beágyazó környezetek közvetlenül is lehetővé teszik a modulban definiált függvények meghívását? A válasz erre kettős: Egyfelől nincsen rá szükség, a CryptID.wasm önmagában is felhasználható. Másfelől viszont, a CryptID-et elsősorban olyan beágyazó környezetekben történő felhasználásra szánjuk, melyek JavaScripten keresztül teszik lehetővé a WebAssembly modulok meghívását. Ilyen környezet például a böngésző vagy a Node.js. Ezekben a környezetekben nagy segítséget jelent, hogy a CryptID ugyanúgy hívható, mint bármely más, JavaScriptben írt könyvtár, a WebAssembly pedig csupán implementációs részlet marad.

E megközelítés támogatásához, még a JavaScript interfész alatt megtalálható egy úgynevezett interoperabilitási (röviden: interop) réteg, melynek egyik fele C-ben, másik fele pedig JavaScriptben készült. Az említett réteg a következő feladatokat látja el:

C függvények becsomagolása. Ahhoz, hogy a C-ben írt függvények JavaScriptből hívhatók legyenek, be kell csomagolni őket az Emscripten által biztosított `Module` objektum `cwrap` metódusa segítségével. Az interop réteg gondoskodik a WebAssembly-oldalon exportált összes függvény helyes becsomagolásáról.

Adattípusok közötti konverzió. A C-ben készült IBE függvények két olyan adattípust is felhasználnak, melyek konverziót igényelnek:

- Míg a JavaScript-oldal a nagy egészeket karakterláncok formájában kezeli (adott számrendszerben reprezentálva az értéket), addig C-ben a GMP könyvtár `mpz_t` típusa használatos. E két típus között kétirányú konverzióra van szükség (hiszen a nagy egészek kimenetet és bemenetet is jelenthetnek), melyet a GMP `mpz_get_str`[1] és `mpz_init_set_str`[2] függvényeinek felhasználásával valósítottunk meg.
- A titkosítást implementáló C függvény a `CipherTextTuple` típus egy példányát állítja elő, mely nyers bájtsorozatokat is tartalmaz. Habár JavaScriptben az `ArrayBuffer`[3] lehetővé teszi a bináris adatok hatékony kezelését, azonban a továbbítás vagy mentés legtöbbször nem nyers formában történik. Ennek elősegítésére az interop réteg Base64 kódolásúra[4] alakítja a `CipherTextTuple` bináris tartalmát.

[1] `https://gmplib.org/manual/Converting-Integers.html#Converting-Integers`

[2] `https://gmplib.org/manual/Simultaneous-Integer-Init-_0026-Assign.html`

[3] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer`

[4] `https://tools.ietf.org/html/rfc4648#section-4`

Kétirányú adatáramlás biztosítása. A WebAssembly modulok egy vagy több elkülönített, lineáris memóriát használhatnak (*WebAssembly Specification*, 2018). A böngészőben történő beágyazás esetében ezek a memóriaterületek a JavaScript kód által használt memóriától izoláltan kerülnek lefoglalásra. Mivel közvetlenül nem lehetséges az összetett típusok (például a különböző `struct` típusok) paraméterként vagy visszatérési értékként való használata, ezért ezen típusok példányait csak a WebAssembly és a JavaScript által birtokolt memóriaterületek közötti másolással lehet kicserélni. Ennek megfelelő lebonyolítása magában foglalja a lineáris memória JavaScript oldalról történő írását és olvasását, valamint a C oldalon történő helyes memóriafoglalást és -felszabadítást.

5.2.5. JavaScript interfész

A JavaScript interfész jelenti a CryptID külvilág számára nyilvános függvényeit és adattípusait. Míg a megelőző rétegek implementációs részletnek tekinthetők, addig erre az interfészre a kliensek közvetlenül is támaszkodhatnak.

Felelősségi körök

Azon felül, hogy az interfész kapcsolatot jelent a könyvtár alsóbb rétegei és az integráló kliens között, további felelősségi köröket is ellát:

A bemenet ellenőrzése. Mivel ez a réteg választja el a könyvtár megvalósítását a külvilágtól, ez az egyetlen olyan pont, ahol hibás adatok léphetnek be a rendszerbe. Ilyen hibás bemenetre számtalan példa adható, elég ha a `null` értékekre, a nem megfelelő alakú objektumokra vagy a hibásan kódolt adatokra gondolunk. Az ilyen rendellenes inputok kiszűrése ebben a rétegben történik, mielőtt még bármilyen számítás alapját képezhetnék.

Kulcskonverzió. A CryptID jelentette újdonságok közül az egyik a strukturált, metaadatokat tartalmazó publikus kulcs, melyet a könyvtár egy JavaScript objektum formájában vár. Fontos követelmény azonban, hogy a tartalmilag azonos JavaScript objektumokból mindig azonos bitsorozatot kell előállítani.

Naiv megoldása lehetne ennek a problémának a következő: először a `JSON.stringify` függvény segítségével karakterláncot képzünk az objektumból, majd ezt egyszerű bitsorozatként interpretáljuk (azaz `char[]` típusként C-ben). E megközelítés azonban tartalmilag azonos objektumok esetén eltérő karakterláncokat eredményezhet, abból kifolyólag, hogy a `JSON.stringify` az objektumok kulcsait azok hozzáadásának sorrendjében konvertálja (Ecma International, 2015).

Ezt megkerülendő, a `JSON.stringify` meghívása előtt a CryptID a publikus kulcsként szolgáló objektumból létrehoz egy vele tartalmilag azonos objektumot, melyhez azonban a kulcsok betűrendben kerülnek hozzáadásra. E konverziónak köszönhetően a JavaScript interfész alatti rétegek már egy bitsorozatot kapnak publikus kulcsként, a strukturáltsággal tehát egyáltalán nem kell foglalkozniuk.

Exportált típusok

A JavaScript interfész által exportált összes típus WebIDL[5] definíciója megtalálható a CryptID – WebIDL definíciók függelékben, a tényleges funkcionalitást biztosító két interfész azonban a könnyebb áttekinthetőség kedvéért az 5.1. kódrészletben is megtalálható.

```
1  interface CryptIDFactory {
2      Promise<CryptID> getInstance();
3  }
4
5  interface CryptID {
6      SetupResult setup(SecurityLevel securityLevel);
7
8      EncryptResult encrypt(PublicParameters publicParameters,
9                            Identity identity,
10                           DOMString message);
11
12     ExtractResult extract(PublicParameters publicParameters,
13                           BigIntegerString masterSecret,
14                           Identity identity)
15
16     DecryptResult decrypt(PublicParameters publicParameters
17                           PrivateKey privateKey,
18                           CipherTextTuple ciphertext);
19
20     void dispose();
21 };
```

5.1. Kódrészlet. A funkcionalitást biztosító interfészek.

Az integráló kód számára a könyvtár a `CryptIDFactory` interfész példányaként jelenik meg. Ezen interfész egyetlen függvénye az objektumgyárként (*factory*) funkcionáló `getInstance`, mely a `CryptID` példányok aszinkron létrehozására képes. A *factory* metódus által biztosított indirekció lehetővé teszi, hogy a visszaadott interfész mögé más és más implementáció kerüljön; például WebAssemblyt nem támogató böngésző esetén egy asm.js megvalósítás. Az aszinkronitás pedig annak enged utat, hogy a hívó kód futása a példány előállítása közben is folytatódhasson. E módon lehetséges például a WebAs-

[5] `https://www.w3.org/TR/WebIDL-1/`

sembly kód lusta (*lazy*) betöltése: a Wasm bináris nem kerül letöltésre a weboldal többi részével együtt, hanem csak a `getInstance` függvény meghívásakor indul el a háttérben a letöltés, feldolgozás és validálás.

Az IBE-hez kapcsolódó rutinokat a `CryptID` interfészen találjuk. E függvények visszatérési értéke minden esetben tartalmaz egy `success` nevű mezőt, mely a kívánt művelet sikerességét jelzi. Ha például a `decrypt` metódust nem a megfelelő privát kulccsal hívjuk, akkor a visszaadott objektum `success` mezője `false` értéket fog felvenni. Azért választottuk ezt a megoldást a kivételekkel szemben (és valójában mellett), mert a kivételeket valóban a kivételes esetek számára tartjuk fent (Bloch, 2008), mint például a hibás bemeneti formátumok, a megsértett invariánsok vagy a váratlan számítási hibák. Úgy gondoljuk, hogy e módon egy könnyebben integrálható, kényelmesebb felületet tudunk biztosítani a kliensek számára.

Érdemes még megemlíteni a `dispose` függvényt, mely a `CryptID` interfész által lefoglalt erőforrások elengedésére, felszabadítására szolgál. Ilyen erőforrás lehet például a WebAssembly implementáció esetén a modul által használt memóriaterület. Ha már nincs szükségünk IBE rutinokra, akkor érdemes a `dispose` segítségével a lehető leghamarabb felszabadítani a nem használt erőforrásokat. A `dispose` hívását követően bármelyik, az adott példány által biztosított függvény meghívása (beleértve magát a `dispose`-t is) hibának számít, és kivételt eredményez.

5.3. Alkalmazásfejlesztés CryptID-del

A következőkben a könyvtár alapszintű felhasználását mutatjuk be egy JavaScript program formájában. Ennek célja, hogy illusztrálja a CryptID integrálásának egyszerű voltát. A könnyebb olvashatóság érdekében a programot kisebb kódrészletekre bontva ismertetjük, a CryptID – Példaprogram függelékben azonban egyben is megtalálható a teljes forráskód.

Előbb a biztosított funkcionalitás rövid leírását adjuk, amit aztán a forráskód és az ahhoz fűzött magyarázatok követnek.

5.3.1. Funkcionalitás

A példaprogram által megvalósított funkcionalitás a következő: Először új nyilvános paramétereket és mesterkulcsot generál. Ezt követően egy előre rögzített üzenetet titkosít egy email címet tartalmazó nyilvános kulccsal. A titkosítás után előállítja a nyilvános kulcshoz tartozó privát kulcsot, végül pedig ennek használatával visszaállítja az eredeti üzenetet a titkos szövegből.

5.3.2. Implementáció

Szeretnénk felhívni a figyelmet arra, hogy az olvashatóság megőrzése érdekében a következő kódrészletek nem tartalmaznak hibakezelést. Ezen felül kiemelnénk, hogy a sorok számozása folytonos, azaz a mindenkori sorszám a teljes forrásban elfoglalt pozíciót tükrözi.

```
1 const { performance } = require('perf_hooks');
2 const CryptIDFactory = require('cryptid');
```

Az első teendőt a felhasznált könyvtárak importálása jelenti. A `perf_hooks` a futási idő mérésére szolgáló rutinokat biztosít, míg a a `cryptid` a korábban említett `CryptIDFactory` interfész egy példányát exportálja.

```
4 const MESSAGE = "Two hashes walk into a bar, one was a salted.";
5 const PUBLIC_KEY = {
6     email: 'bob@example.com'
7 };
```

Mind az üzenet, mind a publikus kulcs rögzített értékkel bír. A publikus kulcs egy JavaScript objektum, mely metaadatokat ezúttal nem, csupán egy azonosításra használható email címet tartalmaz. Ennek valódi szerepe ugyanakkor jelenleg nincsen, hiszen az azonosító ellenőrzése nem része a példaprogramnak.

```
9 (async function main() {
10     const CryptID = await CryptIDFactory.getInstance();
```

Az IBE függvények a `CryptID` interfészen találhatók, melynek példányait a `CryptIDFactory` interfész által definiált `getInstance` metódus képes előállítani. Ennek végrehajtása aszinkron, ezért a JavaScript `async-await` szintaxisát használjuk a visszatérési érték kinyeréséhez.

```
12     console.log(`Original message: "${MESSAGE}"`);
13
14     const start = performance.now();
15     const { publicParameters, masterSecret } = CryptID.setup('lowest');
```

Amennyiben még nem rendelkezünk publikus paraméterekkel és mesterkulccsal, akkor le kell generálnunk őket. Erre szolgál a `setup` függvény, melynek egyetlen paramétere a titkosítás erősségét határozza meg. Ezúttal a `lowest` beállítást választjuk, mely gyenge titkosítást, ugyanakkor rendkívül gyors futást garantál.

Az alkalmazás az IBE rutinok végrehajtási idejét is leméri a rutinok futása előtt és után

rögzített időbélyegek segítségével. Az előbbi időbélyeget a `setup` függvényt megelőzően hozzuk létre, majd eltároljuk a `start` változóban.

```
17      const { ciphertext } = CryptID.encrypt(publicParameters, PUBLIC_KEY, MESSAGE);
```

Publikus paraméterek birtokában már lehetséges az üzenet letitkosítása. Ehhez még szükséges egy nyilvános kulcs is, melyet jelen esetben a `PUBLIC_KEY` változó tárol. Sikeres végrehajtás esetén a visszaadott objektum `ciphertext` mezője tartalmazza a titkos szöveget.

```
19      const { privateKey } = CryptID.extract(publicParameters, masterSecret, PUBLIC_KEY);
```

A titkos szöveg visszafejtése csak egy privát kulcs birtokában lehetséges, melyet az `extract` függvény hívásával generálhatunk. Fontos kihangsúlyozni, hogy a nyilvános kulcsban szereplő adatok ellenőrzését ez a függvény nem végzi el, arról a kliensnek kell gondoskodnia.

```
21      const { plaintext } = CryptID.decrypt(publicParameters, privateKey, ciphertext);
22      const runningTime = performance.now() - start;
```

Az eredeti üzenet visszaállításának utolsó lépését a titkos szöveg visszafejtése jelenti. Ez csak akkor lehet sikeres, ha az üzenet titkosításához használt nyilvános kulcs privát párját használjuk. Mivel a példaprogramban ez biztosan teljesül, ezért a `decrypt` hívása az eredeti üzenettel fog visszatérni, melyet a `plaintext` változóban tárolunk.

A végrehajtási idő mérésének végét jelentő időbélyeg rögzítése is ekkor történik meg. A `runningTime` változó a futás alatt eltelt ezredmásodperceket fogja tartalmazni.

```
24      CryptID.dispose();
25
26      console.log(`Decrypted message: "${plaintext}"`);
27      console.log(`Completed in ${runningTime} ms`);
28  })();
```

A forráskód utolsó soraiban a lefoglalt erőforrások felszabadítását, valamint a visszafejtett üzenet és a futási idő kiíratását találjuk. Habár a `dispose` meghívása jelen esetben nem feltétlenül szükséges (hiszen az alkalmazás végrehajtása után a lefoglalt erőforrások automatikusan felszabadulnak), azonban böngészőben, vagy hosszabb futásra tervezett szerveroldali alkalmazásban kritikus lehet az erőforrásokkal való megfelelő gazdálkodás.

5.4. Teljesítmény

A fejezet megelőző részeiben előbb megismerkedhettünk a könyvtár struktúrájával és implementációs részleteivel, majd a gyakorlathoz közelítve láthattuk az alapszintű használatát is. Mielőtt azonban bemutatnánk a CryptID segítségével építhető nagyobb léptékű alkalmazásokat, szeretnénk a könyvtár teljesítményét is vázolni.

A nemfunkcionális követelmények közül a teljesítmény, azon belül is a végrehajtási idő kritikus szereppel bír. Hiába rendelkezik ugyanis egy megoldás magas biztonsági garanciákkal, ha futási ideje olyan rossz, hogy a gyakorlatban használhatatlan.

A következőkben különböző mérések útján a CryptID legfontosabb komponenseinek teljesítményét elemezzük, minden esetben kitérve az optimalizációs lehetőségekre is. A mérésekhez használt környezetek pontos jellemzői megtalálhatók a CryptID – Teljesítmény függelékben, csakúgy mint a használt metodológia, valamint a pontos mért értékek.

5.4.1. Pont skaláris szorzása

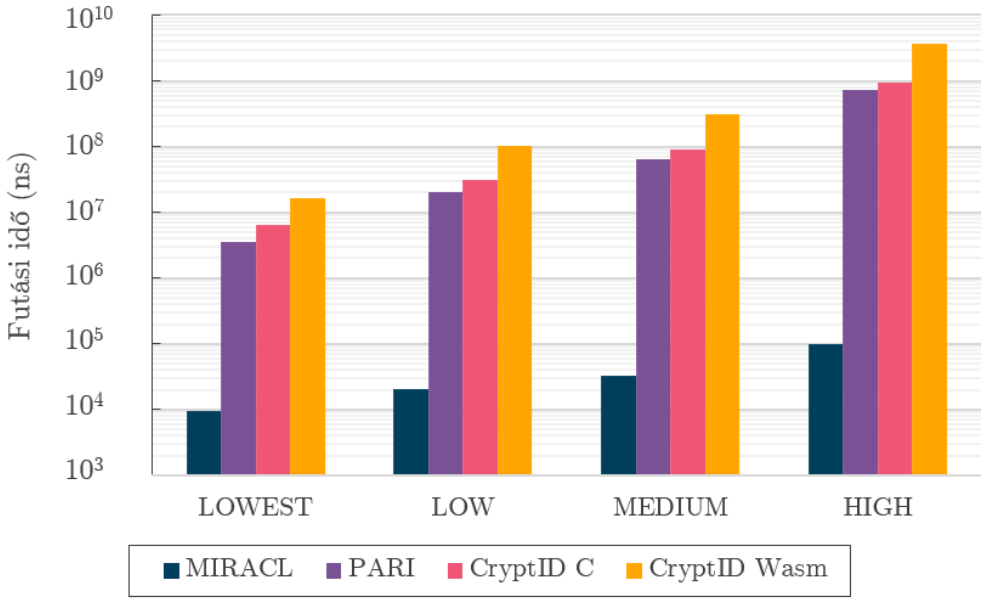
Az IBE-hez szükséges elliptikus görbe aritmetika központi eleme a skaláris szorzás. Ez a művelet számos könyvtárban megtalálható, melyek közül a korábban is említett MIRACL-t és a PARI-t választottuk ki a CryptID-del történő összehasonlításhoz. Összesen tehát négy implementációt hasonlítottunk össze: a natívan futó, C-ben írt MIRACL, PARI és CryptID megoldásokat, valamint a CryptID WebAssembly változatát.

Mivel mind a MIRACL, mind a PARI régóta fejlesztett, hatékony könyvtárnak tekinthető, ezért azt vártuk, hogy ezek biztosan gyorsabbak lesznek még a CryptID C változatánál is. Úgy gondoltuk, hogy az utolsó helyen a WebAssembly verzió fog állni.

Az 5.2. ábrán logaritmikus skálára vetítve láthatjuk az egyes megoldások futási idejét különböző méretű bemenetekre. Az adatokból leolvasható, hogy az elvárásaink javarészt beigazolódtak: a MIRACL minden bemenet esetén nagyságrendekkel gyorsabban fut, mint a többi megoldás, és valóban a WebAssembly végrehajtása tart a legtovább. A MIRACL kitűnő teljesítményét többek között az implementált *wNAF* algoritmus hatékonyságának köszönheti[6] (Özcan, 2006), míg a WebAssembly lomhaságát feltehetőleg a szabvány fiatal volta okozta optimalizációs hiányosságok eredményezhetik. Utóbbi azt jelenti, hogy a WebAssembly verzió a legkisebb bemenet esetén két és félszer, a legnagyobb esetén pedig megközelítőleg négyszer annyi ideig fut, mint a natív bináris.

Kiemelendő ugyanakkor, hogy a PARI és a CryptID C teljesítményében nincs jelentős eltérés. Ennek hátterében az állhat, hogy a CryptID-hez hasonlóan a PARI is a GMP könyvtárra épül, valamint ugyanazt az algoritmust, az úgynevezett *Double-and-Add* mód-

[6] `http://bit.ly/miracl-ecurve-mult`



5.2. ábra. Az elliptikus skaláris szorzás futási ideje.

szert implementálja[7] a skaláris szorzáshoz.

Érdemes azt is kihangsúlyozni, hogy ugyan a CryptID WebAssembly teljesítménye meg sem közelíti a MIRACL eredményét, azonban ennek esetében sem beszélhetünk elviselhetetlenül lassú futásról: `LOWEST` bemenetre ez 0,016, míg `HIGH` bemenetre 3,732 másodpercet jelent.

Optimalizációs lehetőségek A CryptID skaláris szorzás tekintetében nyújtott teljesítménye javítható a MIRACL által is használt *wNAF* algoritmus megvalósításával. Ennek segítségével a könyvtár a futási időt tekintve feltehetően a MIRACL és a PARI között helyezkedne el. Habár a fejlesztést már megkezdtük, jelenleg még nem áll rendelkezésünkre tesztelhető kód.

További sebességnövekedés érhető el ezen felül a Heuberger és Mazzoli által leírt szorzási eljárás implementálásával (2014). A *wNAF* elkészítését követően ezt is ki szeretnénk próbálni.

5.4.2. Tate párosítás

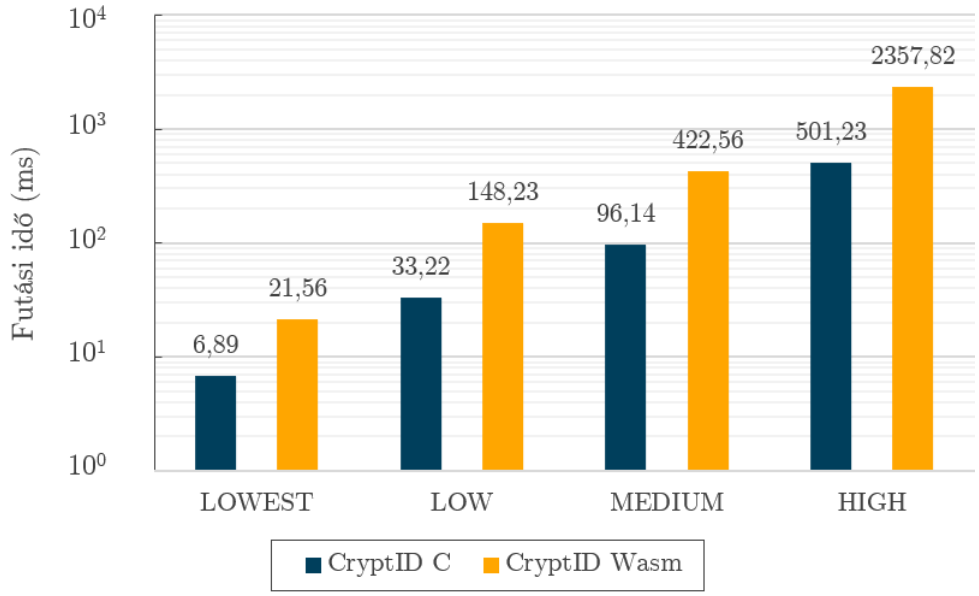
A skaláris szorzás mellett az IBE megvalósítások hangsúlyos eleme valamilyen párosítás. Ahhoz, hogy a titkosítás és a visszafejtés minél gyorsabban kerüljön végrehajtásra, elengedhetetlen ennek hatékony megvalósítása.

Sajnos a párosítást tekintve csak a CryptID C és WebAssembly változatát tudtuk

[7] `http://bit.ly/pari-fpe-mul`

összehasonlítani. Ennek oka, hogy a CryptID az RFC 5091-ben leírt, úgynevezett módosított Tate párosításra épül, mely a már említett könyvtárak egyikében sem található meg közvetlenül.

Felhasználva az előző mérés során szerzett tapasztalatokat, azt vártuk, hogy a WebAssembly változat háromszor-négyszer lassabb lesz, mint a natív.



5.3. ábra. A Tate párosítás futási ideje.

Az 5.3. ábra logaritmikus skálára vetítve mutatja a Tate párosítás különböző nagyságú bemeneteken mért futási idejét. Itt szeretnénk felhívni a figyelmet arra, hogy a grafikonon szereplő adatok ezúttal milliszekundumban kerültek rögzítésre. Az egyes sávok fölött található értékekből könnyen kiszámolható, hogy a WebAssembly bináris megközelítőleg háromszor-ötször tovább fut, mint a natív kód. A különbség tehát a skaláris szorzásnál mérthez hasonló.

Önmagában az ilyen mértékű sebességkülönbség jelenléte kellemetlennek tekinthető, azonban az eltérés konzisztens volta fontos bizonyíték arra vonatkozóan, hogy a WebAssembly környezet teljesítménye stabil, nem ingadozik.

Optimalizációs lehetőségek Az optimalizációra a Tate párosítás esetén két út kínálkozik: hatékonyabb algoritmus implementálása, illetve a meglevő algoritmus által hívott rutinok javítása. A következőkben az utóbbi lehetőséget tekintjük, azonban későbbi munkánk során szeretnénk algoritmikus módosításokat is megvizsgálni.

Az 5.7. ábrán az *Encrypt* teljesítményét meghatározó két függvény, a `TatePairing` és a `HashToPoint` futási ideje szerepel egy `HIGH` bemenetre. A diagramon szereplő adatokból leolvasható, hogy a Tate párosítás teljesítménye mindenekelőtt a modulo hatványozást

végző `modPow` és a multiplikatív inverzt számító `mulInv` futási idejének függvénye.

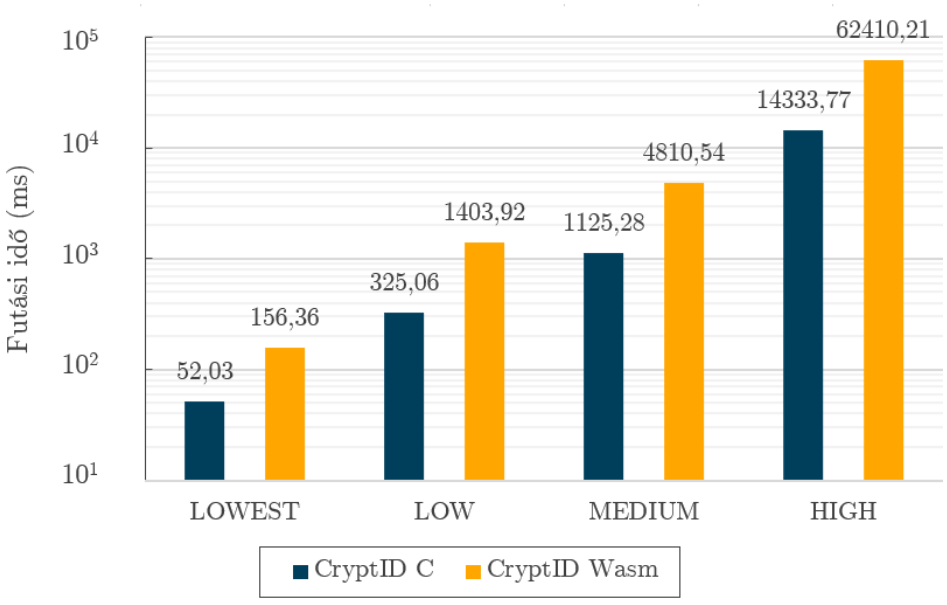
Úgy gondoljuk, hogy mindkét függvény teljesítményében jelentős előrelépés érhető el mikrooptimalizációk alkalmazásával. Ez a processzor *cache* jobb kihasználását, a függvényhívások számának minimalizálását, valamint a függvényparaméterek és visszatérési értékek hatékonyabb kezelését foglalhatja például magában.

5.4.3. Identity-based Encryption

A két legfontosabb alaprutin áttekintése után következhet az Identity-based Encryptiont alkotó függvények (*Setup, Encrypt, Extract, Decrypt*) teljesítményének elemzése. Tekintve, hogy a *Setup* és az *Extract* inkább szerveroldalon kerülhet felhasználásra, ezek esetében csupán a natív C és a Node.js-ben futtatott Wasm verziót hasonlítottuk össze. Az *Encrypt* és a *Decrypt* futási idejét azonban ezeken felül egy asztali és egy mobilos böngészőben is megmértük.

Setup

A *Setup* függvény segítségével új publikus paramétereket és mesterkulcsot állíthatunk elő. Ezek generálása nem tekinthető olyan gyakori tevékenységnek, mint a titkosítás, a visszafejtés vagy a privát kulcs kinyerés. Ennélfogva a *Setup* teljesítményével szemben támasztott elvárásaink kevésbé voltak szigorúak.



5.4. ábra. A *Setup* függvény futási ideje.

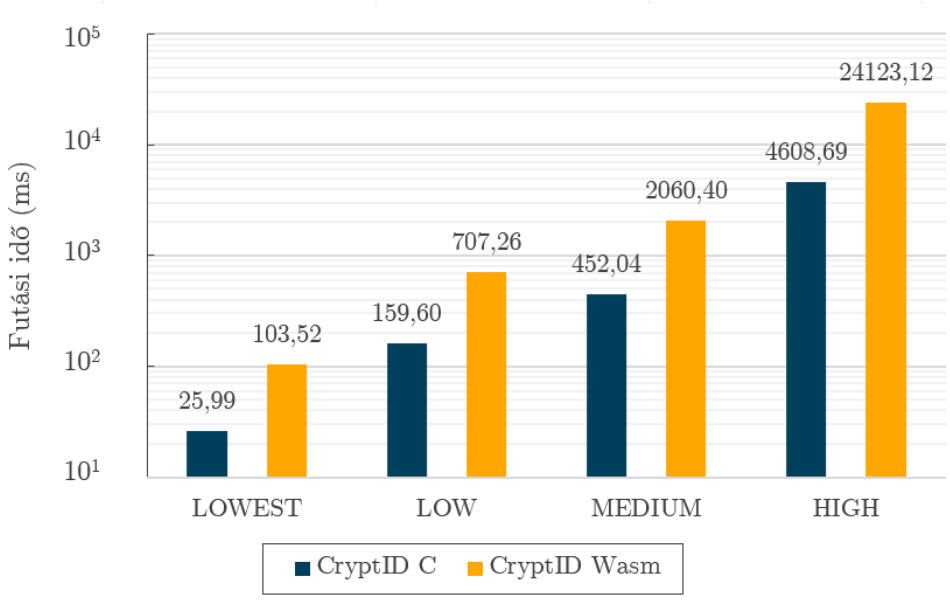
A *Setup* különböző nagyságú bemeneteken mért futási idejét az 5.4. ábrán láthatjuk. Az egyes futási idők már egészen más nagyságrendben mozognak, mint például a skaláris

szorzás esetén: a WebAssembly verzió `HIGH` bementre már több mint egy percig futott! Mi több, ugyanezen bemeneten a C változat is megközelítőleg 15 másodpercig dolgozott.

Optimalizációs lehetőségek A *Setup* teljesítményét a megfelelő értékek generálása határozza meg. Ennek folytán a futási idő javítható a generálási folyamat optimalizálásával, vagy a generált értékek számának csökkentésével. Utóbbi megoldás minimális befektetéssel kínál nagyságrenddekkel jobb teljesítményt, emiatt például a MIRACL is rögzít bizonyos elliptikus görbe paramétereket. Úgy gondoljuk, hogy ez a CryptID számára is megfelelő előrelépési lehetőség lenne, azonban még meg kell határoznunk azokat az értékeket, melyeket a kódban rögzíthetünk.

Extract

Az *Extract* függvény a privát kulcsok kinyerésére szolgál. Ezt a feladatot a PKG végzi, aminek következtében *Setuphoz* hasonlóan az *Extract* esetén is elsősorban szerveroldali felhasználással számoltunk. Alapvető eltérés ugyanakkor, hogy az *Extract* végrehajtási gyakorisága előreláthatólag jóval meghaladja a *Setupnál* feltételezhetőt. Ennek folytán a futási ideje is kellőképpen alacsony kell, hogy legyen.



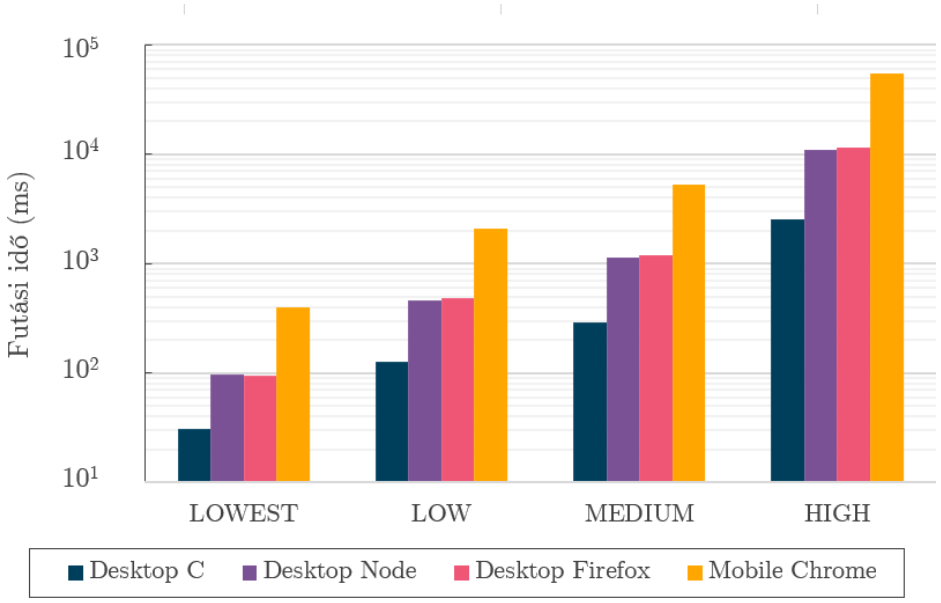
5.5. ábra. Az *Extract* függvény futási ideje.

Az 5.5. ábra a *Extract* függvény futási idejét jeleníti meg. Míg a `LOWEST`, a `LOW` és a `MEDIUM` bemenetek esetén kielégítő értékekről beszélhetünk még a WebAssembly binárist tekintve is (legfeljebb 2 másodperc), addig a `HIGH` bemenethez tartozó eredmények már meghaladják a kívánatosnak gondolt 1–2 másodperces határt. A Wasm verzió különösen rossz, mintegy 24 másodperces eredménnyel rendelkezik.

Optimalizációs lehetőségek Az *Extract* futási idejét befolyásoló egyetlen komponens a `HashToPoint` eljárás. Az 5.7. ábráról leolvasható, hogy ennek teljesítményét a skaláris szorzást megvalósító `affineMul` dominálja. Ez azt jelenti, hogy az *Extract* futási idejét a skaláris szorzás felgyorsításával tudjuk javítani.

Encrypt

A titkosítást megvalósító *Encrypt* függvény futását már egy asztali, valamint egy mobilos böngészőben is lemértük. Ennek oka, hogy a titkosítás kifejezetten egy kliensoldali műveletnek tekinthető, és ennek megfelelően legtöbbször a böngészőben kerül majd végrehajtásra. A hatékony implementáció ezúttal kiemelt fontosságot nyer, hiszen a mobil eszközök csak korlátozott erőforrásokkal rendelkeznek.



5.6. ábra. Az *Encrypt* függvény futási ideje.

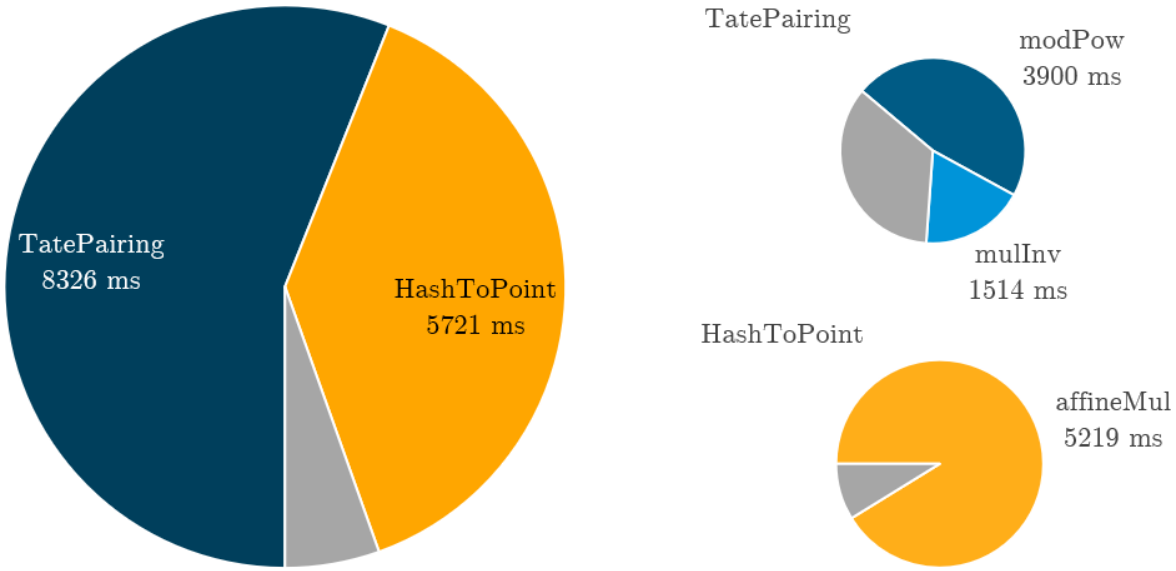
A 5.6. ábrán ismét megfigyelhető a már látott tendencia: a Node.js-alapú WebAssembly háromszor-négyszer nagyobb értékeket produkál, mint a natív kód. Megjelenik emellett az asztali Firefox böngésző, mely nagy meglepetésünkre a Node.js-szel megegyező idők alatt futtatja a függvényt. Ez számunkra egy váratlan fordulat volt, hiszen előzetesen azt vártuk, hogy a böngésző, mint összetettebb beágyazó környezet, a Node.js-nél alacsonyabb teljesítményt fog biztosítani.

Sajnos a mobilos böngészőben történő végrehajtás jelentősen leszakad a többi értéktől: mintegy négyszer-ötször lassabb, mint az asztali WebAssembly. Ennek hátterében természetesen az architekturális, valamint erőforrásbeli különbségeket érdemes keresnünk.

Optimalizációs lehetőségek A könyvtárt integráló alkalmazások felhasználói élményét jelentősen befolyásolja a két legtöbbet használt függvény, az *Encrypt* és a *Decrypt* futási ideje. Ennek folytán kiemelt figyelemmel érdemes kezelni az ezekben adódó optimalizációs lehetőségeket.

A 5.7. ábrán az *Encrypt* egy futása során gyűjtött adatokat láthatunk. A bal oldali kördiagram az *Encrypt* által hívott függvények végrehajtási idejét ábrázolja a teljes futási időhöz képest. Erről egyértelműen leolvasható, hogy a teljesítményt a `TatePairing` és a `HashToPoint` eljárás határozza meg. Ezen függvények összetevőit a jobb oldali kördiagramok ábrázolják.

Az *Encrypt* sebessége tehát a Tate párosításnál ismertetett módszerek segítségével, valamint a `HashToPoint` függvényben szerepet játszó skaláris szorzás felgyorsításával növelhető.

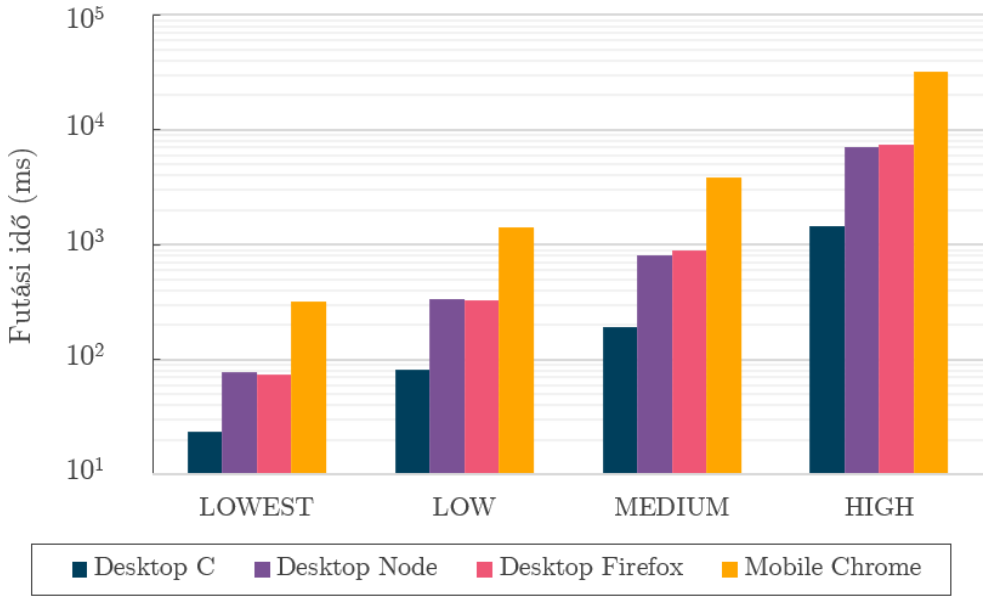


5.7. ábra. Az *Encrypt* függvényt alkotó komponensek futási ideje egy `HIGH` bemenetre (Desktop Firefox környezet).

Decrypt

Az *Encrypthez* hasonlóan a *Decrypt* futási idejét is négy környezetben mértük. Elvárásainkat az előző mérések vezérelték: úgy gondoltuk, hogy a natív C változat lesz a leggyorsabb, azonos időt ad a Desktop Node.js és a Firefox, a sort pedig a mobilos Chrome zárja.

Az 5.8. ábráról leolvasható adatok pontosan az említett elvárásokat igazolják: az *Encryptnél* látott relatív különbségek jelennek meg itt is. Felfedezhető a hasonlóság mindamellett az abszolút futási időt tekintve is: a *Decrypt* végrehajtási ideje megközelítőleg $\frac{2}{3}$-a az *Encryptének*. Ez a párhuzam abból adódik, hogy a *Decrypt* sebességét szinte kizárólag



5.8. ábra. A *Decrypt* függvény futási ideje.

a `TatePairing` határozza meg, `HashToPoint` hívás viszont ezúttal nincsen.

Optimalizációs lehetőségek A *Decrypt* teljesítményét a Tate párosítás dominálja, így ennek javítása tekinthető a legígéretesebb optimalizációs lehetőségnek.

5.4.4. Összegzés

A CryptID alapjául szolgáló legfontosabb rutinok teljesítményét látva kijelenthető, hogy asztali környezetben a könyvtár használhatónak tekinthető még nagyobb bemenet esetén is, mobil eszközökön azonban csak kisebb bemenetekre várhatunk kielégítő végrehajtási időt. Érdemes azonban ezen értékeléshez hozzátenni, hogy ezt a teljesítményt a CryptID javarészt egyszerűbb algoritmusok kevésbé hatékony megvalósításaival éri el. Emiatt számos, az előzőekben is felsorolt optimalizációs lehetőség adódik, melyek alkalmazásával a könyvtár teljesítménye jelentősen növelhető, megfelelő futási időt nyújtva mobil eszközökön is.

6. fejezet

Alkalmazások

Az 5.3. alfejezet API-használatra fókuszáló példaprogramja után ebben a fejezetben nagyobb léptékű alkalmazásokat mutatunk be. A fejezet célja, hogy valós, gyakorlati példákat adjon olyan webalkalmazásokra, melyek kulcsösszetevője lehet az IBE, s ezáltal a CryptID.

Előbb egy ténylegesen implementált webalkalmazást – CrpytID.email – ismertetünk, majd Személyre szabott zárthelyi néven egy rövid esettanulmány formájában vázoljuk az IBE egy lehetséges egyetemi felhasználását.

6.1. Implementáció – CryptID.email

A CryptID.email egy fájltitkosításra szolgáló platformfüggetlen webalkalmazás, melynek legfontosabb jellemzői a következők:

Email, mint azonosító. Az IBE-t használó alkalmazások implementációjában kulcsfontosságú kérdés, hogy mit használjunk azonosítóként. Míg egy vállalati belső alkalmazás esetén ez a legtöbbször adott (például LDAP útján), addig egy nyílt, tetszőleges felhasználó által igénybe vehető szolgáltatás esetén ez korántsem egyértelmű.

Az azonosító kiválasztásának két kiemelkedő szempontja az azonosító elterjedtsége (hány felhasználó rendelkezik vele?) valamint ellenőrizhetősége (mennyire bonyolult ellenőrizni, hogy egy azonosító valóban egy adott entitáshoz tartozik?). Ezek figyelembevételével a CrpytID.email esetén az email címet választottuk azonosítóként, hiszen az internetfelhasználók zöme rendelkezik legalább egy email címmel, valamint ellenőrizni is egyszerűbb, mint egy telefonszámot vagy egy Facebook-hozzáférést.

Szimmetrikus tartalomtitkosítás. Az aszimmetrikus módszerek (mint az IBE) alacsony teljesítményük miatt nem alkalmasak nagyobb adattömeg titkosítására. Emi-

att gyakori megoldás, hogy az aszimmetrikus kriptográfiát egy nagyságrendekkel gyorsabb szimmetrikus módszer kulcsának titkosítására használják.

Ezen ötlet jelenik meg a CryptID.email implementációjában is: Először generálunk egy véletlenszerű bájtsorozatot, mely a fájlok tartalmának titkosításához használt AES (*Advanced Encryption Standard*) eljárás kulcsa lesz. Az AES titkosítás végeztével pedig az említett kulcsot a CryptID könyvtárral aszimmmetrikusan titkosítjuk.

Ennek köszönhetően egyszerre élvezhetjük az IBE nyújtotta előnyöket, valamint a szimmetrikus titkosítás gyorsaságát.

Kliensoldali titkosítás. Az IBE meghatározó jellemzője, hogy a visszafejtésre alkalmas privát kulcsokat egy megbízható fél, a PKG generálja. Ez azonban azt is jelenti, hogy a PKG tetszőleges publikus kulcshoz képes privát kulcsot előállítani. Ugyanakkor ez a lehetőség csak akkor jelent kockázatot, ha a PKG (vagy valamely, azzal közvetlen kapcsolatban álló komponens) a publikus kulcson kívül az azzal titkosított adatokhoz is hozzáfér.

Ennek elkerülésére a CryptID.email kliensoldali titkosítást valósít meg, azaz mind a titkosítás, mind a visszafejtés úgy történik, hogy a publikus kulcson kívül semmilyen adat nem hagyja el a felhasználó eszközét.

A következőkben a titkosítás, majd a visszafejtés lépésein keresztül ismertetjük az alkalmazás megvalósításának részleteit.

6.1.1. Titkosítás

A fájl beolvasása

A titkosítási folyamat első lépése – természetesen a megfelelő webhely felkeresését követően – a titkosítandó fájl beolvasása. Ez a fájl teljes tartalmának memóriába történő beolvasását jelenti. Ugyan a CryptID.email által használt FileReader API[1] lehetővé teszi, hogy egyszerre csak a fájl egy kisebb szeletét olvassuk be és dolgozzuk fel (azaz *streameljük*), azonban a böngészők jelenleg nem biztosítanak hatékony *streaming* titkosítási eljárásokat, így ennek megfelelő kihasználása nem lehetséges.

A fájl tartalmának titkosítása

Miután a memóriában rendelkezésre áll a titkosítandó adat, a CryptID.email a WebCrypto API-n[2] keresztül előbb generál egy AES kulcsot, majd ugyanezen API-t használva AES-

[1] `https://w3c.github.io/FileAPI/#dfn-filereader` – `https://caniuse.com/#feat=filereader`
[2] `https://www.w3.org/TR/WebCryptoAPI/` – `https://caniuse.com/#feat=cryptography`

GCM (*Galois/Counter Mode*) blokktitkosítással rejtjelezi a bemenetet.

Ezzel a memóriában már készen áll egy szimmetrikus privát kulcs, valamint az ezzel titkosított adattömeg.

Címzett kiválasztása

Az IBE titkosítás megkezdése előtt a felhasználónak meg kell adnia a címzett email címét, ami a publikus kulcsban szereplő azonosítót jelenti.

Publikus paraméterek lekérése

A szimmetrikus kulcs titkosítása előtt szükséges a publikus IBE paraméterek beszerzése. Ezeket a szerver szolgáltatja, mely a paramétereken felül egy azokhoz köthető azonosítót is elküld a kliensnek. Erre azért van szükség, mert a szerver havonta új publikus paramétereket és mesterkulcsot generál (a `CryptID.setup` függvényt igénybe véve), viszont a korábban titkosított adatok feloldására ezek természetesen nem alkalmasak, így tárolni és azonosítani kell őket.

A szimmetrikus kulcs titkosítása

A következő lépés a fájl tartalmának rejtjelezéséhez használt szimmetrikus kulcs titkosítása. Ez a CryptID könyvtár `encrypt` függvényén keresztül történik, az előző lépés során megszerzett publikus paraméterek és az email címből képzett publikus kulcs segítségével.

A titkos fájl összeállítása

A CryptID.email a titkosított tartalmat nem juttatja el a címzetthez – arról a felhasználónak kell gondoskodnia. Ennek elősegítésére a CryptID.email létrehoz egy titkos fájlt, mely a következő összetevőket foglalja magában:

Eredeti fájlnév. A titkosított fájlhoz tartozó eredeti fájlnévre elsősorban a kiterjesztés megőrzése miatt van szükség, mely számos operációs rendszeren kitüntetett jelentéssel bír.

Publikus paraméterek. A szimmetrikus kulcs titkosításához felhasznált publikus IBE paraméterek azonosítója. A CryptID.email szerveroldali komponense ez alapján tudja megkeresni a privát kulcs generálásához szükség mesterkulcsot.

Titkosított szimmetrikus kulcs. Az IBE-titkosított AES kulcs.

Titkosított fájltartalom. Az eredeti fájl tartalma AES-titkosítva.

A titkos fájlt ezután a felhasználó a memóriából a háttértárra mentheti, majd igény szerint továbbíthatja.

6.1.2. Visszafejtés

A visszafejtés lépéssora ott veszi fel a fonalat, hogy a felhasználó valamilyen módon hozzájutott egy, a CryptID.emailen keresztül előállított titkos fájlhoz, és szeretné visszafejteni az ebben tartalmazott eredeti fájlt. A visszafejtés azonban csak egy megfelelő privát kulcs birtokában sikerülhet. A privát kulcs beszerzésehez pedig a felhasználónak egy azonosítási folyamaton keresztül igazolnia kell az email címét.

Tekintve, hogy a visszafejtési folyamat a titkosítás inverze, megjelennek hasonló lépések, mint például a fájl memóriába olvasása. Ezek részletes ismertetését itt elhagyjuk, és csak a lényeges pontokat mutatjuk be részletesen.

Email cím megadása

Az azonosítási folyamat elindításához a felhasználónak először meg kell adnia az email címét a CryptID.email weboldalán.

Megerősítő email kiküldése

A CryptID.email szerveroldali komponense ezután generál egy nyolc karakter hosszúságú, úgynevezett email tokent, melyet SendGriden[3] keresztül kiküld a felhasználó által megadott email címre. A token csupán tíz percig érvényes, ezt követően nem alkalmas az email cím igazolására.

Email token megadása

Ha a megadott email cím valóban a felhasználó tulajdonában áll, akkor megkapja az említett email tokent is, melyet aztán megadhat a CryptID.email weboldalán. Ezzel a felhasználó igazolta az identitását, elindulhat a privát kulcs kinyerése.

Privát kulcs generálás

Az identitás ellenőrzését követően a szerver meghatározza a fájl titkosítása során használt publikus paramétereket, valamint a privát kulcs generálásához szükséges mesterkulcsot a titkos fájlban tárolt azonosítón keresztül. Ezeket a korábban megadott email cím mellett továbbítja a PKG felé, mely a `CryptID.extract` függvényt meghívva előállítja a publikus kulcs privát párját. Végül a szerver a privát kulcsot elküldi a kliensnek.

[3] `https://sendgrid.com/`

A szimmetrikus kulcs visszafejtése

A privát kulcs birtokában a kliens megpróbálkozhat a titkos fájlban tárolt szimmetrikus kulcs visszafejtésével. Ehhez a kliens a CrpytID könyvtár `decrypt` függvényét használja. Amennyiben a privát kulcs helyes, azaz valóban a szóban forgó felhasználó email címével titkosították a szimmetrikus kulcsot, akkor a visszafejtés sikeres lesz. Ellenkező esetben a folyamat itt megáll, a felhasználó nem fér hozzá az eredeti tartalomhoz.

Az eredeti fájl visszaállítása

A folyamat utolsó lépése az eredeti fájl tartalmának visszafejtése, mely az előzőleg nyert szimmetrikus kulcs és a WebCrypto API segítségével történik. A memóriában előálló nyílt reprezentációt ezután a felhasználó a háttértárra mentheti. A visszafejtési folyamat ezzel véget ért, visszaállítottuk az eredeti fájlt.

6.1.3. Összegzés

A CryptID.email különböző eszközökön és platformokon átívelő egységes titkosítási megoldást nyújt. A használatához nem szükséges sem előzetes kulcscsere, sem a PKI (*public key infrastructure*) használata, hiszen a továbbítandó fájlok titkosításához csupán a címzett email címét kell megadnunk.

6.2. Esettanulmány – Személyre szabott zárthelyi

Míg az előző alkalmazás a CryptID hordozható és könnyen használható voltát demonstrálta, addig a személyre szabott zárthelyit leíró rövid esettanulmány elsődleges célja a metaadatok beágyazásának bemutatása. Tapasztalataink szerint a metaadatok leginkább egy adott szakterületen belül lehetnek hasznosak, ezért kézenfekvő választás volt az esettanulmány témájának az általunk jól ismert zárthelyi dolgozatok világa.

Az alkalmazás hallgatónként egyedi elektronikus feladatsorok összeállítását teszi lehetővé, melyek a titkosításnak köszönhetően akár napokkal a dolgozat időpontja előtt elhelyezhetők az egyetemi számítógépeken.

6.2.1. Motiváció

A motiváció a csalások megelőzésében gyökerezik: úgy gondoljuk, hogy a hálózati hozzáférés korlátozásával szemben sokkal hatásosabb eszközt jelent a visszaélések megakadályozásában, ha minden hallgató eltérő feladatsorral rendelkezik. Ennek kezelése azonban a különböző feladatsorok előállításán és ellenőrzésén felül további problémákat is felvet.

Ahhoz, hogy ez az elképzelés valóban csökkentse a csalások számát, biztosítani kell, hogy minden hallgató csak a saját feladatsorához férjen hozzá és csak a zárthelyi időpontjában.

Úgy gondoltuk, hogy erre a célra a CryptID biztosította IBE egy valódi megoldást nyújthat.

6.2.2. Megvalósítás

A rendszer elméleti megvalósítását a publikus kulcs megtervezésén keresztül mutatjuk be. A következőkben egyenként hozzáadunk három mezőt a publikus kulcshoz, kifejtve egyúttal az adott mező implementációval való kapcsolatát. A publikus kulcs tartalmának ilyen módon történő leírása teret enged az alkalmazás összes fontos aspektusának bemutatására.

Neptun-kód

A publikus kulcs nélkülözhetetlen összetevője egy, a hallgatók azonosítására szolgáló mező. Természetes választás a Neptun-kód, mellyel Karunk minden hallgatója rendelkezik. Előnyös tulajdonsága továbbá, hogy ellenőrzése már kiépített infrastruktúrán keresztül megtehető: a privát kulcs generálása előtt a hallgató a hálózati azonosítóját használva bejelentkezik, ezzel igazolva az identitását.

Önmagában mindazonáltal a Neptun-kód nem elégséges. Ha a feladatsort csak a címzett hallgató azonosítójával titkosítjuk, akkor a visszafejtéshez használt privát kulcs tetszőleges, a hallgatónak szánt feladatsor visszafejtésére alkalmas lesz!

Dátum

Érdemes tehát az előbb említett probléma megakadályozására egy újabb mezővel, a dátummal kiegészíteni a publikus kulcsot. Titkosításkor a Neptun-kód mellett a zárthelyi dolgozat dátumát is felhasználjuk, a privát kulcs generálása előtt pedig a rendszer automatikusan elhelyezi az aktuális dátumot a publikus kulcsban. Ez rendkívül egyszerűen megtehető, hiszen csak egy JSON objektumot kell egy újabb mezővel kiegészíteni.

Gondot jelenthet ugyanakkor, amennyiben a hallgató egy napon több zárthelyit is ír. Erre egy lehetséges válasz a publikus kulcsban elhelyezett dátum finomságának növelése, például *év-hónap-nap* helyett *év-hónap-nap-óra* mezők használatával. Jobb megoldást kínál azonban a publikus kulcs további bővítése.

Tárgy

A bővítés ötlete azon alapszik, hogy a titkosításhoz felhasználhatjuk a szóban forgó tárgy nevét is. Nem szabad azonban elfelejteni, hogy csak olyan mezőt érdemes elhelyeznünk a

publikus kulcsban, melyet aztán a privát kulcs generálásakor ellenőrizni tudunk.

A *tárgy* mezőt ezért a dátumhoz hasonlóan közvetlenül a szerveroldalon helyezzük el a publikus kulcsban, a privát kulcs generálása előtt, megkerülve ezzel az ellenőrzés szükségességét. Ekkor azonban biztosítani kell, hogy a mező értéke megbízható forrásból származzék. Ehhez a hallgató által a zárthelyi megírásához használt számítógép IP címe nyújthat alapot. Ha az IP cím alapján meghatározható a terem, ahol az eszköz elhelyezkedik, akkor ezt az információt a teremfoglalási adatbázissal összekapcsolva egyértelműen meg tudjuk mondani, hogy a hallgató épp milyen tárgyhoz kötődő zárthelyin vesz részt.

Ez a mező tehát a fizikai hely közvetett ellenőrzésének lehetőségét nyújtja.

6.2.3. Összegzés

A vázolt alkalmazás garantálja, hogy az egyes feladatsorok visszafejtése csak a kívánt hallgató által, a kívánt időben és helyen végezhető el. Mindezen követelmények kikényszerítése automatikusan, a publikus kulcsban elhelyezett metaadatok segítségével történik.

Ennek köszönhetően a dolgozatokhoz használt feladatsorok jóval a zárthelyi tényleges időpontja előtt elhelyezhetők a számítógépeken, valamint akár hálózati korlátozás is alkalmazható – csupán a PKG-hez történő hozzáférést kell biztosítani.

Mindamellett nem szabad elfelejtkeznünk arról, hogy CryptID alternatívája lehet az egyszerű szimmetrikus titkosítás alkalmazása: a feladatsorok előállításával egy időben minden feladatsorhoz generálunk egy szimmetrikus kulcsot, melyhez a hallgató csak sikeres azonosítás után férhet hozzá. Csakhogy az IBE egyik lehetséges alkalmazási területét épp az ilyen, sok szimmetrikus kulcs menedzselését igénylő rendszerek kiváltása jelenti, hiszen a mesterkulcson felül semmilyen más kulcs tárolására nincsen szükség.

7. fejezet

Összefoglalás

A dolgozatunkban a Boneh-Franklin IBE egy újszerű megvalósítását mutattuk be, előbb a matematikai alapokat, aztán az implementáció részleteit kifejtve. Az elkészült könyvtár célja, hogy az elérhető megvalósítások valós alternatívája legyen, egyedi jellemzőinek köszönhetően.

7.1. A CryptID jellemzői

A CryptID újdonságtartalommal bíró sajátosságai két irányból is megközelíthetők, előbb a technológiai alapokat, majd az IBE rendszert tekintve. A könyvtár egyedülálló karakterisztikája a hordozhatóság: a WebAssembly biztosította platformfüggetlenségre építve asztali, mobil és IoT eszközökön is elérhető titkosítási szolgáltatást nyújt. A hordozhatóság egyszerű integrálhatósággal párosul, mely megkönnyíti a dolgozatban ismertetetthez hasonló webes vagy asztali alkalmazások elkészítését.

Hangsúlyos a publikus kulcs szerepe is: az IBE egyedülálló tulajdonsága, hogy a publikus kulcs valójában egy azonosító; ezt bővíti ki a CryptID metaadatok hozzáadásal, ráadásul mindezt strukturált, könnyen kezelhető formában. Ez egészen új szakterület-specifikus alkalmazások előtt nyitja meg az utat, melyre egy szemléletes példa a 6.2. alfejezetben bemutatott személyre szabott zárthelyi.

A funkcionalitás mellett nagy figyelmet fordítottunk az implementáció megfelelő teljesítményére is, hiszen a kriptográfiai programkönyvtárak összehasonlításának egyik kiemelt szempontja a különböző erőforrásokkal (memória, processzoridő) való hatékony gazdálkodás. Ez még hangsúlyosabb szerepet nyer, ha a korlátozott erőforrásokkal rendelkező mobil eszközöket tekintjük. A 5.4. alfejezetben elemzett mérések tanúsága szerint a CryptID kielégítő futási idővel rendelkezik asztali és mobil eszközökön is.

7.2. Továbblépési lehetőségek

A könyvtár további fejlesztése számos irányban folytatható. Kiemelkedik ezek közül azonban a korábban részletezett optimalizációk által kijelölt irány, melynek mentén a CryptID futási ideje nagyságrendekkel csökkenthető. Ilyen például a Heuberger-Mazzoli elliptikus skaláris szorzás, a rögzített paraméterek használata, vagy a hatékonyabb Tate párosítás alkalmazása. Ezen felül teljesítménynövekedést érhetünk el különböző mikrooptimalizációk segítségével is.

Nemcsak a futási idő, hanem a bináris méretének csökkentése is fontos szempont, hiszen ezzel mérsékelhető a hálózati adatforgalom nagysága. Tekintve, hogy a CryptID méretének jelentős részét a külső könyvtárak (GMP, OpenSSL) teszik ki, így szeretnénk megvizsgálni ezek eltávolításának, kiváltásának vagy tömörítésének lehetőségét. Utóbbira hatékony megoldást jelenthet például az úgynevezett *tree-shaking*, mely a nem használt kódrészletek eldobását jelenti.

Annak érdekében, hogy a CryptID valós alkalmazások alapját szolgáltathassa, növelnünk kell az ismertségét, melyet különböző konferenciákon történő részvétellel szeretnénk elősegíteni. Ezek egyúttal kiváló alkalmat kínálnak a visszajelzések, fejlesztési lehetőségek gyűjtésére is.

A titkosításon túl további azonosító alapú rendszerek is léteznek, amelyek magja rendkívül hasonló. Ilyenek például az Identity-based Signature (Yi, 2003) és Identity-based Cloud Authentication Protocol (Huszti & Oláh, 2018). A hasonlóság miatt megfontolandó az efféle rendszerek implementációja is, hiszen a CryptID magában hordoz számos olyan réteget, amelyek a további sémák fejlesztése során módosítás nélkül vagy kis módosítással újrahasználhatók.

Irodalomjegyzék

Adobe Flash Player. (2018). Letöltve, 2018. 10. 12.: https://www.adobe.com/hu/products/flashplayer.html

Bernstein, D. J. & Lange, T. (2008). *Analysis and optimization of elliptic-curve single-scalar multiplication.* American Mathematical Society. Letöltve: https://doi.org/10.1090/conm/461/08979 doi: 10.1090/conm/461/08979

Bloch, J. (2008). *Effective Java (2nd Edition) (The Java Series)* (2nd ed.). Upper Saddle River, NJ, USA: Prentice Hall PTR.

Boneh, D. & Franklin, M. K. (2001). Identity-Based Encryption from the Weil Pairing. *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology* (pp. 213–229). London, UK, UK: Springer-Verlag. Letöltve: http://dl.acm.org/citation.cfm?id=646766.704155

Boyen, X. & Martin, L. (2007). Identity-based cryptography standard (IBCS) #1: Supersingular curve implementations of the BF and BB1 cryptosystems. *RFC*, *5091*, 1–63. Letöltve: https://doi.org/10.17487/RFC5091 doi: 10.17487/RFC5091

A break from the past. (2015). Letöltve, 2018. 10. 12.: https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/

Buchmann, J. A., Karatsiolis, E. & Wiesmaier, A. (2013). *Introduction to public key infrastructures.* Springer Publishing Company, Incorporated.

Can I use WebAssembly? (2018). Letöltve, 2018. 10. 13.: https://caniuse.com/#feat=wasm

The Case for Elliptic Curve Cryptography. (2009). Letöltve, 2018. 10. 16.: https://web.archive.org/web/20090117023500/http://www.nsa.gov/business/programs/elliptic_curve.shtml

Clark, L. (2017a). *A crash course in just-in-time (JIT) compilers.* Letöltve, 2018. 10. 13.: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/

Clark, L. (2017b). *What makes WebAssembly fast?* Letöltve, 2018. 10. 13.: https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/

Daniel Ehrenberg (Szerk.). (2018a). *WebAssembly JavaScript Interface.* Letöltve, 2018. 10. 13.: https://webassembly.github.io/spec/js-api/index.html

Daniel Ehrenberg (Szerk.). (2018b). *WebAssembly Web API.* Letöltve, 2018. 10. 13.: https://webassembly.github.io/spec/web-api/index.html

Designing Secure ActiveX Controls. (2017). Letöltve, 2018. 10. 12.: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa752035%28v%3dvs.85%29

Digital Signature Standard (DSS) (Tech. Rep.). (2013, jul). Letöltve: https://doi.org/10.6028/nist.fips.186-4 doi: 10.6028/nist.fips.186-4

Ecma International (Szerk.). (2015). *ECMAScript 2015 Language Specification.* Letöltve, 2018. 10. 17.: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf

Emit WebAssembly by default instead of asm.js. (2018). Letöltve, 2018. 10. 13.: https://github.com/kripken/emscripten/issues/6168

Flash & The Future of Interactive Content. (2017). Letöltve, 2018. 10. 12.: https://theblog.adobe.com/adobe-flash-update/

Friedman, D. P. & Wand, M. (2008). *Essentials of programming languages, 3rd edition.* The MIT Press.

Gay, J. (2001). *The History of Flash.* Letöltve, 2018. 10. 12.: https://web.archive.org/web/20090202103237/http://www.adobe.com:80/macromedia/events/john_gay/index.html

Granlund, T. & the GMP development team. (2016). *GNU MP: The GNU Multiple Precision Arithmetic Library* (6.1.2 ed.). Letöltve, 2018. 10. 16.: http://gmplib.org/

Grimes, R. A. (2001). *Malicious Mobile Code: Virus Protection for Windows.* O'Reilly Media.

Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., … Bastien, J. (2017, jun). Bringing the web up to speed with webassembly. *SIGPLAN Not.*, *52*(6), 185–200. Letöltve: http://doi.acm.org/10.1145/3140587.3062363 doi: 10.1145/3140587.3062363

Hankerson, D., Menezes, A. & Vanstone, S. (2004). *Guide to Elliptic Curve Cryptography.* Springer-Verlag New York.

Heuberger, C. & Mazzoli, M. (2014, aug). Symmetric digit sets for elliptic curve scalar multiplication without precomputation. *Theoretical Computer Science*, *547*, 18–33. Letöltve: https://doi.org/10.1016/j.tcs.2014.06.010 doi: 10.1016/j.tcs.2014.06.010

Horner, W. G. (1819, July). A new method of solving numerical equations of all orders,

by continuous approximation. *Philosophical Transactions of the Royal Society of London*, *109*, 308–335. Letöltve: https://www.jstor.org/stable/107508

Huszti, A. & Oláh, N. (2018). Identity-Based Cloud Authentication Protocol. *The 11th Conference of PhD Students in Computer Science* (pp. 33–36). University of Szeged, Institute of Informatics. Letöltve: http://www.inf.u-szeged.hu/~cscs/pdf/cscs2018.pdf

JavaDocs: java.applet.Applet. (2018). Letöltve, 2018. 10. 12.: https://docs.oracle.com/javase/8/docs/api/java/applet/Applet.html

Java Plug-In Technology. (2018). Letöltve, 2018. 10. 12.: https://www.oracle.com/technetwork/java/index-jsp-141438.html

The Java Tutorials: Java Applets. (2018). Letöltve, 2018. 10. 12.: https://docs.oracle.com/javase/tutorial/deployment/applet/index.html

JEP-289. (2017). Letöltve, 2018. 10. 12.: http://openjdk.java.net/jeps/289

Joux, A. (2000). A One Round Protocol for Tripartite Diffie-Hellman. *Lecture Notes in Computer Science* (pp. 385–393). Springer Berlin Heidelberg. Letöltve: https://doi.org/10.1007/10722028_23 doi: 10.1007/10722028_23

Koblitz, N. (1987, January). Elliptic curve cryptosystems. *Mathematics of Computation*, *48*(177), 203–209. Letöltve: https://www.jstor.org/stable/2007884 doi: 10.2307/2007884

Kovács, G. (2014). *Személyre szabott titkosítási rendszerek megvalósítása* (Diplomamunka, Debreceni Egyetem, Debrecen, Hungary). Letöltve: http://hdl.handle.net/2437/193181

Lang, S. (1978). *Elliptic Curves: Diophantine Analysis.* Springer-Verlag Berlin Heidelberg.

List of languages that compile to JS. (2018). Letöltve, 2018. 10. 13.: https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS

Lynn, B. (2007). *On the implementation of pairing-based cryptosystems* (Doctoral dissertation, Stanford University, Stanford, CA, USA). Letöltve: https://crypto.stanford.edu/pbc/thesis.pdf

Martin, L. (2008). *Introduction to Identity-based Encryption.* Boston: Artech House.

Menezes, A., Vanstone, S. & Okamoto, T. (1991). Reducing elliptic curve logarithms to logarithms in a finite field. *Proceedings of the twenty-third annual ACM symposium on Theory of computing - STOC '91.* ACM Press. Letöltve: https://doi.org/10.1145/103418.103434 doi: 10.1145/103418.103434

Microsoft Announces ActiveX Technologies. (1996). Letöltve, 2018. 10. 12.: https://news.microsoft.com/1996/03/12/microsoft-announces-activex-technologies/

Miller, V. S. (1985). Use of elliptic curves in cryptography. *Lecture Notes in Computer Science* (pp. 417–426). Yorktown Heights NY, USA: Springer-Verlag Berlin Heidelberg. Letöltve: https://doi.org/10.1007/3-540-39799-X_31 doi: 10.1007/3-540-39799-X_31

Mrabet, N. E. & Joye, M. (2016). *Guide to Pairing-Based Cryptography.* Chapman & Hall/CRC.

NaCl and PNaCl. (2017). Letöltve, 2018. 10. 13.: https://developer.chrome.com/native-client/nacl-and-pnacl

Native Client: Technical Overview. (2017). Letöltve, 2018. 10. 13.: https://developer.chrome.com/native-client/overview

Nelson, B. (2017). *Goodbye PNaCl, Hello WebAssembly!* Letöltve, 2018. 10. 13.: https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html

Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the Internet. (1995). Letöltve, 2018. 10. 12.: https://tech-insider.org/java/research/1995/1204.html

Nyakacska, L. (2016). *Elliptikus görbék és bilineáris párosítások* (Diplomamunka, Debreceni Egyetem, Debrecen, Hungary). Letöltve: http://hdl.handle.net/2437/232828

OpenSSL Cryptography and SSL/TLS Toolkit (1.1.0h ed.). (2018). Letöltve, 2018. 10. 16.: https://www.openssl.org

Özcan, A. B. (2006). *Performance analysis of elliptic curve multiplication algorithms for elliptic curve cryptography* (Diplomamunka, Middle East Technical University, Ankara, Turkey). Letöltve: https://etd.lib.metu.edu.tr/upload/12607698/index.pdf

PARI/GP version 2.11.0. (2018). Univ. Bordeaux. Letöltve, 2018. 10. 16.: http://pari.math.u-bordeaux.fr/

Shamir, A. (1985). Identity-based Cryptosystems and Signature Schemes. *Proceedings of CRYPTO 84 on Advances in Cryptology* (pp. 47–53). New York, NY, USA: Springer-Verlag New York, Inc. Letöltve: http://dl.acm.org/citation.cfm?id=19478.19483

WebAssembly Specification. (2018). Letöltve, 2018. 10. 13.: https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

Wheeler, D. A. (2015). *Secure Programming HOWTO.* Letöltve, 2018. 10. 16.: https://dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., … Fullagar, N. (2009). Native client: A sandbox for portable, untrusted x86 native code. *Proceedings of the 2009 30th ieee symposium on security and privacy* (pp. 79–93). Washington, DC,

USA: IEEE Computer Society. Letöltve: https://doi.org/10.1109/SP.2009.25 doi: 10.1109/SP.2009.25

Yi, X. (2003, feb). An identity-based signature scheme from the weil pairing. *IEEE Communications Letters*, *7*(2), 76–78. Letöltve: https://doi.org/10.1109/lcomm.2002.808397 doi: 10.1109/lcomm.2002.808397

Zakai, A. (2015). *Big Web App? Compile It!* Letöltve, 2018. 10. 13.: kripken.github.io/mloc_emscripten_talk/

Zakai, A. (2017). *Why WebAssembly is Faster Than asm.js.* Letöltve, 2018. 10. 13.: https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/

Függelék

CryptID – WebIDL definíciók

```
interface CryptIDFactory {
    Promise<CryptID> getInstance();
}

interface CryptID {
    SetupResult setup(SecurityLevel securityLevel);

    EncryptResult encrypt(PublicParameters publicParameters,
                          Identity identity,
                          DOMString message);

    ExtractResult extract(PublicParameters publicParameters,
                          BigIntegerString masterSecret,
                          Identity identity)

    DecryptResult decrypt(PublicParameters publicParameters
                          PrivateKey privateKey,
                          CipherTextTuple ciphertext);

    void dispose();
};

typedef Point PrivateKey;

typedef DOMString Base64String;

typedef DOMString BigIntegerString;

typedef object Identity;

enum SecurityLevel { "lowest", "low", "medium", "high", "highest" };

dictionary Point {
    BigIntegerString x;
    BigIntegerString y;
};

dictionary PublicParameters {
    BigIntegerString fieldOrder;
```

```
40      BigIntegerString subgroupOrder;
41      Point pointP;
42      Point pointPpublic;
43      SecurityLevel securityLevel;
44  };
45
46  dictionary CipherTextTuple {
47      Point cipherU;
48      Base64String cipherV;
49      Base64String cipherW;
50  };
51
52  dictionary SetupResult {
53      boolean success;
54      BigIntegerString? masterSecret;
55      PublicParameters? publicParameters;
56  };
57
58  dictionary EncryptResult {
59      boolean success;
60      CipherTextTuple? ciphertext;
61  };
62
63  dictionary ExtractResult {
64      boolean success;
65      PrivateKey? privateKey;
66  };
67
68  dictionary DecryptResult {
69      boolean success;
70      DOMString? plaintext;
71  };
```

CryptID – Példaprogram

```
const { performance } = require('perf_hooks');
const CryptIDFactory = require('cryptid');

const MESSAGE = "Two hashes walk into a bar, one was a salted.";
const PUBLIC_KEY = {
    email: 'bob@example.com'
};

(async function main() {
    const CryptID = await CryptIDFactory.getInstance();

    console.log(`Original message: "${MESSAGE}"`);

    const start = performance.now();
    const { publicParameters, masterSecret } = CryptID.setup('lowest');

    const { ciphertext } = CryptID.encrypt(publicParameters, PUBLIC_KEY, MESSAGE);

    const { privateKey } = CryptID.extract(publicParameters, masterSecret, PUBLIC_KEY);

    const { plaintext } = CryptID.decrypt(publicParameters, privateKey, ciphertext);
    const runningTime = performance.now() - start;

    CryptID.dispose();

    console.log(`Decrypted message: "${plaintext}"`);
    console.log(`Completed in ${runningTime} ms`);
})();
```

CryptID – Teljesítmény

A mérésekhez használt környezetek

Paraméter	Érték
Típus	Dell Inspiron 5567 (2017)
Processzor	i7-7500U, 2,7 GHz
Memória	16 GB, DDR4, 2133 MHz

7.1. táblázat. Desktop hardverkonfiguráció.

Szoftver	Verzió
Operációs rendszer	Ubuntu 16.04.4 LTS
emscripten	1.38.8
gcc	5.4.0 20160609
Node.js	v8.9.1

7.2. táblázat. Desktop natív és Node.js szoftverkonfiguráció.

Szoftver	Verzió
Operációs rendszer	Windows 10 Education, Version 1803 (OS Build 17134.35)
Firefox Quantum	62.0.3

7.3. táblázat. Desktop Firefox szoftverkonfiguráció.

Paraméter	Érték
Típus	Nokia 6.1 TA-1043
Processzor	Qualcomm Snapdragon 630, 2.2 GHz
Memória	3 GB, LPDDR4, 1333 MHz

7.4. táblázat. Mobil hardverkonfiguráció.

Szoftver	Verzió
Operációs rendszer	Android 8.1.0 - Kernel 4.4.78-perf+
Chrome for Mobile	68.0.3440.91

7.5. táblázat. Mobil szoftverkonfiguráció.

Könyvtár	Verzió
libpari-dev	2.11.0-1
MIRACL	commit 4bd13901519d329c6c1369fb3322b52fe10c7a6e[1]

7.6. táblázat. A felhasznált könyvtárak verziói.

Mérési metodológia

A teljesítménymérésekhez minden platformon a Google Benchmark[2] könyvtárat használtuk.

Egy teszt egy adott függvény adott bementre történő n egymást követő lefuttatását jelenti (ezeket iterációknak nevezzük), ahol n értékét a Benchmark könyvtár határozza meg a függvény sebességének megfelelően. A teszt eredménye az iterációk során mért idők átlaga.

Az egyes függvényekhez rendelt végleges időket függvényenként húsz, a fentieknek megfelelő teszt eredményének átlagából számoltuk.

Azért választottuk az átlagidőt a futási idő jellemzésére, mert a teszteredmények elemzése során úgy láttuk, hogy ez írja le legjobban az mérésekben rejlő mintázatokat, valamint szemantikát tekintve is ezt gondoltuk a felhasználói élménnyel leginkább párhuzamba hozható metrikának.

Mérési eredmények

	MIRACL	PARI	CryptID C	CryptID WebAssembly
LOWEST	$9{,}32 \times 10^3$ ns	$3{,}55 \times 10^6$ ns	$6{,}33 \times 10^6$ ns	$1{,}61 \times 10^7$ ns
LOW	$2{,}02 \times 10^4$ ns	$2{,}03 \times 10^7$ ns	$3{,}16 \times 10^7$ ns	$1{,}01 \times 10^8$ ns
MEDIUM	$3{,}22 \times 10^4$ ns	$6{,}49 \times 10^7$ ns	$8{,}94 \times 10^7$ ns	$3{,}06 \times 10^8$ ns
HIGH	$1{,}01 \times 10^5$ ns	$7{,}32 \times 10^8$ ns	$9{,}55 \times 10^8$ ns	$3{,}73 \times 10^9$ ns

7.7. táblázat. A skaláris szorzás futási ideje.

[2] `https://github.com/google/benchmark`, verzió: commit af441fc1143e33e539ceec4df67c2d95ac2bf5f8

	CryptID C	CryptID WebAssembly
LOWEST	$6{,}89 \times 10^6$ ns	$2{,}16 \times 10^7$ ns
LOW	$3{,}32 \times 10^7$ ns	$1{,}48 \times 10^8$ ns
MEDIUM	$9{,}61 \times 10^7$ ns	$4{,}23 \times 10^8$ ns
HIGH	$5{,}01 \times 10^8$ ns	$2{,}36 \times 10^9$ ns

7.8. táblázat. A Tate párosítás futási ideje.

	CryptID C	CryptID WebAssembly
LOWEST	52,03 ms	156,36 ms
LOW	325,06 ms	1403,92 ms
MEDIUM	1125,28 ms	4810,54 ms
HIGH	14 333,77 ms	62 410,21 ms

7.9. táblázat. A *setup* futási ideje

	CryptID C	CryptID WebAssembly
LOWEST	25,9884 ms	103,5176 ms
LOW	159,6002 ms	707,2616 ms
MEDIUM	452,0390 ms	2060,4003 ms
HIGH	4608,6934 ms	24 123,1172 ms

7.10. táblázat. Az *extract* futási ideje.

	Natív C	Node.js Wasm	Desktop Firefox Wasm	Mobil Chrome Wasm
LOWEST	30,6817 ms	97,2830 ms	93,7125 ms	400,9150 ms
LOW	128,3681 ms	460,7209 ms	484,1000 ms	2088,7400 ms
MEDIUM	291,9348 ms	1140,2769 ms	1190,8000 ms	5277,2400 ms
HIGH	2554,0214 ms	11 108,9214 ms	11 422,1000 ms	55 161,8300 ms

7.11. táblázat. Az *encrypt* futási ideje.

	Natív C	Node.js Wasm	Desktop Firefox Wasm	Mobil Chrome Wasm
LOWEST	23,6065 ms	78,1959 ms	73,9000 ms	322,1050 ms
LOW	82,7325 ms	335,8183 ms	327,5500 ms	1432,9000 ms
MEDIUM	193,3273 ms	805,8475 ms	881,4000 ms	3869,7500 ms
HIGH	1469,1147 ms	7112,7990 ms	7397 ms	32 273,5100 ms

7.12. táblázat. A *decrypt* futási ideje.

Saját munka leírása

Bagossy Attila

- Implementáltam a CryptID.ref Tate párosítását, valamint kialakítottam a publikus API-t.
- Elkészítettem a CryptID.wasm és a CryptID.js közötti interoperabilitási réteget, illetve implementáltam a CryptID.js-t.
- Teljesítménymméréseket készítettem, melyek mind asztali, mind mobil eszközön demonstrálják a könyvtárat alkotó legfontosabb függvények futási idejét.
- Megvalósítottam a CryptID.email alkalmazást, mely egy gyakorlati példán keresztül mutatja be a CryptID használatát.

Vécsi Ádám

- A CryptID.ref könyvtárban elkészítettem a következőket:
  - komplex számokon értelmezett műveletek,
  - elliptikus görbe aritmetika,
  - a Boneh-Franklin IBE függvényei.
- A referencia-implementációt alapul véve a fentieket, illetve a Tate párosítást implementáltam a CryptID.wasm részeként is, integrálva a GMP és OpenSSL könyvtárakat.
- Kidolgoztam a személyre a szabott zárthelyi alkalmazás ötletét.