Debreceni Egyetem
Informatikai Kar

Az RSA algoritmus hatékony implementációja C++ és assembly nyelveken

Témavezető:
Dr. Fazekas Gábor
egyetemi docens

Készítette:
Varga Csaba
programtervező matematikus

Debrecen
2008

Tartalomjegyzék

1. Bevezetés

1.1. A probléma felvázolása

A kriptográfia (szó szerinti fordításban "rejtett írás") célja az információ átalakítása oly módon, hogy azt egy adott személy vagy csoport továbbra is értelmezni tudja, de a kívülállók számára érthetetlenné váljon. Alkalmazása valószínűleg egyidős magával az írással – már az ókori Mezopotámiából maradt ránk olyan agyagtábla, amelyre titkosírással rögzítettek recepteket. A múlt század végéig csak kevesek használták, főleg állami, katonai vagy üzleti titkok védelmére. Ez mára megváltozott, a kriptográfia életünk részévé vált: amikor mobiltelefonálunk, bankkártyával vásárlunk vagy csak belépünk a számítógépünkbe, a háttérben kriptográfiai algoritmusok gondoskodnak arról, hogy illetéktelenek ne hallgathassanak le minket, ne ismerjék meg banki adatainkat, és ne tudják meg jelszavunkat. Éppen ezért fontos, hogy gyors, ugyanakkor biztonságos algoritmusaink legyenek, és ezeket hatékonyan tudjuk implementálni a meglevő eszközeinken.

Az RSA az egyik legkorábbi nyilvános kulcsú algoritmus, amely a mai napig is biztonságos (megfelelően hosszú kulcs esetén). Éppen ezért rengeteg helyen használják, így fontos a hatékony implementációja. Célom egy ilyen implementáció elkészítése volt a lehető legkevesebb külső kód felhasználásával, demonstrálva az ehhez szükséges hosszúszám algoritmusokat. Dolgozatomban bemutatom ezeket az algoritmusokat, illetve megindoklom, miért ezeket választottam a rendelkezésre állók közül.

1.2. Alkalmazott technológiák

A program x86 illetve x86-64 processzorokon, Microsoft Windows operációs rendszeren fut. A forráskód nagy része C++ nyelven készült, mely magas szintű platformfüggetlen nyelv, de lehetővé teszi a hardverközeli programozást a teljesítményigényes részeknél. Assembly nyelven valósítottam meg a leginkább teljesítményigényes algoritmusokat, illetve azokat, amelyek hatékony megvalósításához a C++-ból közvetlenül nem elérhető primitív művelet szükséges. Minden assembly nyelvű kódnak megvan a C++-os megfelelője, amely lehetővé teszi a más platformra való átültetést – igazán jó teljesítmény eléréséhez azonban ezeket az új platformon is gépi szinten kell újraírni.

Az RSA minden számolási lépését saját kód végzi, külső programkönyvtárat csak a véletlenszám-generáláshoz alkalmaztam. (Windows és Linux operációs rendszereken lehetséges a rendszer saját véletlenszám-generátorának használata is.) Bár a megfelelő véletlenszám-generátor létfontosságú az algoritmus biztonságához, működése nem tartozik a hosszúszám-aritmetika témakörébe; a téma méltó tárgyalása magában kitöltene egy dolgozatot. A program használja még a ZThreads nevű programkönyvtárat, amely a

többszálú futtatást teszi lehetővé; segítségével a két processzormaggal rendelkező számítógépeken párhuzamosan futhatnak egyes részszámítások.

1.3. Köszönetnyilvánítás

Szeretnék köszönetet mondani dr. Fazekas Gábor témavezetőmnek, aki hasznos szakirodalmat bocsátott a rendelkezésemre, és tanácsaival segítette a program elkészültét. Szeretném megköszönni továbbá dr. Pethő Atillának, hogy Kriptográfia kurzusán bevezetett ebbe az izgalmas tudományágba.

2. Kriptográfiai alapok

2.1. Alapfogalmak

*Titkosítás*nak nevezzük azt a folyamatot, amikor egy adatsort (a *nyílt szöveg*et) egy adott algoritmust követve és egy másik adatsor, az ún. *kulcs* felhasználásával átalakítjuk az ún. *kriptoszöveg*gé. *Visszafejtés* alatt ennek az ellenkezőjét értjük, amikor egy kriptoszövegből egy algoritmus és kulcs segítségével megkapjuk a nyílt szöveget. Mint látni fogjuk, sem az algoritmusnak, sem a kulcsnak nem feltétlenül kell egyeznie a titkosítás és a visszafejtés során.

A *kriptográfia* célja olyan algoritmusok és kulcsok létrehozása, amelyek a visszafejtést a kulcs ismeretében könnyűvé teszik, míg a kulcs ismerete nélkül a lehető legjobban megnehezítik. A *kriptoanalízis* célja ezzel szemben olyan módszerek kifejlesztése, amelyek a visszafejtést a kulcs ismerete nélkül is a lehető legjobban megkönnyítik. Ha találunk olyan eljárást, amivel egy adott titkosító algoritmus kimenete a kulcs nélkül is elfogadható időn belül visszafejthető, azt mondjuk, hogy az algoritmust *feltörtük*, és a továbbiakban nem tekinthető biztonságosnak a használata. Az „elfogadható idő” kifejezést szándékosan nem konkretizáljuk, ugyanis ez nézőpont kérdése: elvi szempontból az algoritmus fel van törve, ha nem kell az összes lehetséges kulcsot kipróbálnunk a visszafejtéshez; gyakorlati szempontból csak akkor beszélünk feltörésről, ha a jelenleg elérhető hardvereszközökkel hamarabb vissza tudjuk fejteni az adatokat, mint ahogy azok elévülnének. Meg kell még különböztetnünk az algoritmus és egy implementációjának feltörhetőségét: még ha az algoritmusban nem is találtak problémát, az implementáció során akkor is előfordulhat olyan hiba, amely az adott implementációt sebezhetővé teszi.

Egy kriptográfiai algoritmust *szimmetrikus*nak nevezünk, ha a titkosításhoz és a visszafejtéshez használt kulcsok azonosak, vagy egymásból minimális erőfeszítéssel megkaphatók. Azokat a kriptográfiai algoritmusokat, amelyeknél a két kulcs különbözik, és csupán

az egyik ismeretében a másikat nem tudjuk elfogadható idő alatt előállítani, *aszimmetrikusnak* nevezzük.

2.2. Klasszikus algoritmusok

A számítógép feltalálása előtti titkosító algoritmusokat klasszikus algoritmusoknak nevezzük. Ezek mind szimmetrikusak voltak; pusztán emberi erővel lehetetlen lenne elvégezni azokat a számításokat, amelyek az aszimmetrikus algoritmusokhoz szükségesek. A klasszikus algoritmusokat két csoportra bonthatjuk:

- A *helyettesítéses* titkosítás a bemenő szöveg betűit (vagy betűcsoportjait) egyenként képezi le új betűkre (vagy betűcsoportokra). A kulcs egy olyan adatsor, amelyből vissza tudjuk állítani a leképezési szabályt. A legegyszerűbb esetben a leképezési szabály minden betűnél azonos; ekkor *monoalfabetikus* titkosításról beszélünk. Ha a leképezési szabály kódolás közben változik, akkor *polialfabetikus* a titkosítás. A monoalfabetikus és az egyszerűbb polialfabetikus titkosítások könnyen feltörhetők betűstatisztika készítésével: a kriptoszöveg leggyakoribb betűi valószínűleg megfelelnek a nyílt szöveg nyelvének leggyakoribb betűinek. Ha pedig már ismerjük a leggyakoribb betűk eredetijét, a többi megfeleltetés már a szövegkörnyezetből kikövetkeztethető.

- A *permutációs* titkosítás nem változtatja meg a bemenő szöveg betűit, csak azok sorrendjét. A kulcs a sorrend megváltoztatásához használt permutáció. A permutációs titkosítás anagrammák keresésével törhető – ha egy értelmes szó anagrammáját találjuk a kriptoszövegben, akkor feltehetjük, hogy a nyílt szövegben az adott értelmes szó szerepelt, és az anagrammából visszakövetkeztethetünk a felhasznált permutációra.

Mint láthatjuk, mindkét alapelv támadható, főleg, ha számítógépet is használhatunk a támadáshoz. Épp ezért ezeket a módszereket ma már nem használják adatvédelemre.

2.3. Szimmetrikus algoritmusok

Bár magában mind a helyettesítéses, mind a permutációs módszerek támadhatók, érdekes módon a kettő kombinációja segít mindkettejük gyengeségén: a szövegkörnyezet elveszik a permutáció során, így a ritkább karaktereket nehezebb visszafejteni; más részről a helyettesítés elrejti az anagrammákat. Mindezen tovább javíthatunk *törde*lés*sel: az eredeti betűinket kisebb szimbólumok kombinációjára bontjuk, majd ezeken a kisebb szimbólumokon végezzük a titkosítást. A kisebb szimbólumok gyakorisága már nem fog megfelelni

az eredeti nyelv betűgyakoriságának, így a betűstatisztikás támadásnak nem lesz többé haszna. A titkosítás nehézsége tovább növelhető, ha egymás után több permutációs és helyettesítéses lépést hajtunk végre.

Ezzel el is jutottunk a modern szimmetrikus algoritmusokhoz. Egy ember ezeket a lépéseket lassan és sok hibával tudná csak elvégezni, egy számítógép viszont gyorsan és tökéletesen dolgozik. Külön tördelési fázisra nincs is szükség, hiszen a bemenetünk eleve bitekre van tördelve.

A modern szimmetrikus algoritmusok azonban továbbra is rendelkeznek a klasszikus algoritmusok egy fontos problémájával: a kulcsok azonosságával. A kulcsot bizalmasan kell kezelni; ha egy harmadik személy megszerzi, akkor vissza tudja fejteni az adott kulccsal folytatott összes kommunikációt. Ebből következik, hogy meg kell bíznunk a kommunikációs partnerünkben (nem adja ki másnak a kulcsot) valamint a csatornában (senki nem szerzi meg a kulcsot útközben). Ebből következik továbbá, hogy minden partnerünkhöz egyedi kulcsot kell használnunk, hiszen egyébként egyik partnerünk lehallgathatná a másikkal folytatott kommunikációt. A gyakorlatban ezeket a feltételeket nagyon nehéz megvalósítani.

2.4. Aszimmetrikus algoritmusok

A fenti problémára Whitfield Diffie és Martin Hellman találtak megoldást 1976-ban. Az elv lényege, hogy személyenként két kulcsot generálunk: egy *nyilvános kulcs*ot és egy *privát kulcs*ot, úgy, hogy az egyikből a másik kiszámítása ne legyen megoldható. A nyilvános kulcs használható kódolásra, a privát kulcs pedig dekódolásra; a nyilvános kulcsunkat tetszőlegesen terjeszthetjük, a privát kulcsunkat viszont senkinek nem adjuk ki. Ha valaki üzenni akar nekünk, üzenetét a nyilvános kulcsunkkal titkosítja, majd elküldi nekünk. Mivel a visszafejtéshez szükség van a privát kulcsra, amit csak mi ismerünk, biztosak lehetünk benne, hogy az üzenet tartalmát harmadik személy nem ismerte meg.

Ahhoz, hogy ez a séma működjön, olyan leképezést kell találnunk, amelynek az inverze sokkal nehezebben számítható ki, mint maga a leképezés, még akkor is, ha ismerjük a használt (nyilvános) kulcsot. Szükségünk van azonban olyan információra is, amivel az inverz kiszámítása könnyűvé válik; ez az információ lesz a privát kulcs. Az ilyen tulajdonságokkal rendelkező függvényeket *csapóajtó függvény*eknek nevezzük. A klasszikus helyettesítés és permutáció nyilvánvalóan nem ilyenek, ezért az elvnek gyökeresen különböznie kell a szimmetrikus titkosítások elvétől. Sajnos a gyakorlatban nehéz csapóajtó függvényeket találni; a jelenleg használt függvényeinkről úgy gondoljuk, hogy nehéz kiszámítani az inverzüket, de erre nincsen matematikai bizonyíték. Van viszont bizonyítékunk arra, hogy az RSA algoritmus inverze kvantumszámítógépen gyorsan számítható; ha a kvantumszámítógépek elérik a gyakorlati alkalmazhatóság szintjét, az algoritmus nem fog

többé biztonságot nyújtani.

2.5. Hibrid algoritmusok

A csapóajtó függvények általában valamilyen numerikus probléma megoldását igénylik; ez sokkal erőforrás-igényesebb feladat, mint a szimmetrikus titkosítások permutációja és helyettesítése. Emiatt a titkosítás és a visszafejtés is sokkal lassabbak, mint szimmetrikus esetben; általában túl lassúak ahhoz, hogy valós idejű adatforgalomhoz (pl. titkosított webböngészéshez) használni lehessen őket. Ilyen feladatra szimmetrikus titkosítást sem használhatunk, mivel a túloldallal nincs előre megbeszélt kulcsunk, és maga a csatorna lehallgatható. A megoldás a két módszer ötvözése – a szimmetrikus titkosításhoz használt kulcsot véletlenszerűen generáljuk, majd aszimmetrikus titkosítással juttatjuk el a túloldalhoz. Maga a kulcs általában kis méretű, ezért nem jelent gondot aszimmetrikus algoritmussal titkosítani. Az érdemi adatforgalom visszafejtéséhez a támadónak ismernie kellene a szimmetrikus kulcsot, amit viszont csak a címzett privát kulcsával lehet visszafejteni, ezt pedig a címzett titokban tartja. A rendszer biztonságához elengedhetetlen, hogy a szimmetrikus kulcsot megfelelő véletlenszám-generátor szolgáltassa – ha a támadó ki tudja következtetni a következő véletlenszámot, akkor az aszimmetrikus titkosítás feltörése nélkül is hozzáférhet az adatokhoz.

3. Az RSA algoritmus

3.1. Az algoritmus leírása

Az RSA algoritmus nevét kifejlesztőiről, Ron Rivest, Adi Shamir és Leonard Adleman kriptográfusokról kapta, akik 1977-ben publikálták eredményüket. Az algoritmus alapját adó csapóajtó-függvény a moduláris hatványozás, azaz $a^e \bmod m$ kiszámítása, ahol e és m előre megadott értékek (a kulcs részei), a pedig a titkosítandó adat. Jelenlegi ismereteink szerint a moduláris hatványozás inverzét leghatékonyabban akkor lehet kiszámítani, ha ismert m prímtényezős felbontása. Jelenleg nem rendelkezünk viszont olyan algoritmussal, ami a nagy prímtényezőkkel rendelkező számokat elfogadható időn belül (polinom időben) képes lenne prímtényezőire bontani. A faktorizálás annál nehezebb, minél nagyobbak a tényezők, ezért a legerősebb védelmet két, közel azonos nagyságrendű tényezővel érhetjük el. Ha tehát m-et két nagy prím szorzataként állítjuk elő, a moduláris hatványozás tényleg csapóajtó-függvényként viselkedik. A csapóajtót a két prímtényező "nyitja". Ezeket a tényezőket titokban tartjuk, és bízhatunk benne, hogy más nem fogja tudni kiszámítani őket, még úgy sem, hogy m-et nyilvánosságra hozzuk. Az e értékére csak annyi megkötés

van, hogy egynél nagyobb páratlan számnak kell lennie.[1]

Az inverz kiszámítása a következő elven működik: legyenek p és q az m titkos prímtényezői. Keressünk egy olyan d_1 számot, amire igaz az, hogy $ed_1 = k(p-1)+1$ valamilyen k értékre; másképp kifejezve, d_1 legyen olyan, hogy $ed_1 \equiv 1 \pmod{p-1}$. (Ilyen számot könnyen találhatunk a kibővített euklideszi algoritmus segítségével.) A kis Fermat-tétel értelmében $a^{p-1} \equiv 1 \pmod{p}$, ha p nem osztja a-t. Ebből következik, hogy ha p nem osztja a-t, akkor

$$(a^e)^{d_1} \equiv a^{ed_1} \equiv a^{k(p-1)+1} \equiv \left(a^{p-1}\right)^k a \equiv 1^k a \equiv a \pmod{p}$$

Ha viszont p osztja a-t, akkor $a \equiv 0 \pmod{p}$, ezért tetszőleges hatványa is 0 lesz modulo p. Ezzel bebizonyítottuk, hogy minden a-ra $(a^e)^{d_1} \equiv a \pmod{p}$. A bizonyítás során p-ről csak azt használtuk fel, hogy prím, ezért ugyanez a gondolatmenet q-ra is alkalmazható egy d_2 kitevővel. Ha ismerjük $a \bmod p$ és $a \bmod q$ értékét, akkor a kínai maradéktétel értelmében elő tudjuk állítani $a \bmod pq = a \bmod m$ értékét, és ezzel készen is vagyunk. A privát kulcsunk d_1, d_2, p és q értékeiből fog állni.

A gondolatmenetben egy lépéssel tovább is mehetünk. Ha olyan d értéket választunk, amelyre a két prím feltétele egyszerre teljesül ($ed \equiv 1 \pmod{p-1}$ és $ed \equiv 1 \pmod{q-1}$), akkor elegendő egyszer hatványozni a d-vel a fenti két hatványozás helyett: a kapott hatvány kongruens lesz a-val modulo p és modulo q, de mivel ez a két szám relatív prím, az eredmény kongruens lesz a-val modulo pq. Ekkor a privát kulcsunk d és m értékeiből áll.

Mindkét módszernek megvannak az előnyei. A d_1 és d_2 számok jóval kisebbek, mint a d, ezért, mint látni fogjuk, jóval gyorsabb lesz a hatványozás az első módszerrel. A második módszernél viszont a kódolás és a dekódolás ugyanazzal az algoritmussal végezhető; ez hasznos lehet, ha a programkód mérete korlátozott, vagy hardveres megoldás esetén szeretnénk az áramkör bonyolultságát alacsonyan tartani. Az első módszer párhuzamosítható, ha rendelkezésre áll két független számító egység, míg a második nem.

3.2. Támadási lehetőségek

Az RSA algoritmus csak akkor biztonságos, ha megfelelő körültekintéssel alkalmazzuk. A megfelelő implementációhoz szükséges ismernünk az RSA ellen eddig felfedezett támadási lehetőségeket:

- *Faktorizálás nyers erővel*: Ahogy fejlődik a számítási kapacitás, egyre nagyobb szá-

[1] A gyakorlatban e-t nem önkényesen szoktuk választani; olyan számot használunk, amivel gyorsan tudunk hatványozni, hogy minél gyorsabb lehessen a titkosítás. Gyakori választás az $e = 2^{16}+1 = 65537$. Mint látni fogjuk, ezzel a kitevővel a hatványozás csupán 16 négyzetre emelést és egy szorzást igényel. A programom is ezt a kitevőt használja a titkosításhoz.

mok faktorizálása válik kifizetődővé. Ezért fontos, hogy az m hosszát mindig nagyobbra válasszuk, mint amennyit előre láthatólag kifizetődő lesz faktorizálni az elkövetkező pár évben. A jelenleg nyilvánosságra hozott legnagyobb faktorizált érték 663 bit hosszú, és a faktorizáláshoz egy 2.2 GHz-es órajelű AMD Opteron processzornak körülbelül 75 évre lett volna szüksége [Factor-Record]. Ennek ismeretében az 512 bit hosszú kulcsok már nem számítanak biztonságosnak, az 1024 bit hosszú kulcsok pedig csak rövid távra (néhány év) használhatók. Amennyiben a kulcsnak évtizedekig biztonságosnak kell maradnia, 4096 bites vagy annál hosszabb kulcsok használata javasolt.

- *Faktorizálás speciális alak kihasználásával*: Léteznek algoritmusok, amelyek a hagyományos faktorizálásnál gyorsabban lefutnak, ha az m valamilyen speciális alakban van. A Fermat-faktorizáció például nagyon gyorsan felfedi a prímtényezőket, ha $|p - q| < 2 * \sqrt[4]{m}$; Pollard $p - 1$ algoritmusa hatékony, ha $p - 1$ vagy $q - 1$ csak kis prímtényezőket tartalmaz. Az ilyen sebezhetőségek elhárítása érdekében érdemes ellenőrizni a kulcsgenerálás után, hogy a generált kulcs nincs egyik támadható alakban sem. Ennek a támadásnak régebben nagyobb volt a jelentősége; a mai faktorizációs algoritmusok már képesek a fent említett algoritmusok teljesítményére anélkül, hogy bármilyen speciális alakot feltételeznének. A témakört jól körüljárja Rivest és Silverman egyik dolgozata[Rivest-Silverman].

- *Választott nyílt szöveges támadás*: Ha a támadó valamilyen módon képes volt néhány lehetőségre leszűkíteni azt, hogy mi lehetett a nyílt szövegben, akkor a nyilvános kulcs felhasználásával megpróbálhatja titkosítani ezeket a lehetőségeket, és amelyiknek a kódolt alakja egyezik az elfogott titkosított üzenettel, az volt az eredeti üzenet. Ennek elkerülése végett az RSA kódolás előtt ún. *kitöltési sémát* alkalmaznak. Ez egy olyan algoritmus, ami a nyílt szöveghez véletlenszerű „zajt" kever, amelyet a címzett később el tud távolítani. Ez a „zaj" gondoskodik róla, hogy ugyanazt a szöveget kétszer titkosítva két különböző kriptoszöveget kapjunk, és meghiúsítsuk a próbálgatásos módszert. (A kitöltés másik fontos indoka az, hogy az RSA titkosításnak mindig van három fixpontja: a 0, 1 és $m - 1$ értékek mindig önmagukba képződnek le. Kitöltés nélkül a csupa nulla bájtból álló hosszú szakaszok változatlanul jelennének meg a kimenetben, információt szolgáltatva a nyílt szövegről.)

- *Az implementációs részleteket kihasználó támadás*: Léteznek természetesen olyan támadások is, amik nem magát az elvet, hanem annak konkrét implementációját támadják. Ha például lehetőségünk van a visszafejtést végző processzor megfigyelésére, indíthatunk ún. *időzítéses támadás*t (timing attack), amikor is ismert

kriptoszövegek visszafejtésének időtartamából próbálunk következtetni a kulcs bitjeire. Ennek elhárítására használhatunk olyan speciális algoritmusokat, amelyek futásideje kevésbé függ a bemenetektől. A másik megoldás a *vakítás*, amikor a kriptoszöveget megszorozzuk egy r véletlenszám kódolt alakjával, azaz $a^e \bmod m$ helyett $r^e a^e \bmod m$ értékét fejtjük vissza. Ebből $ar \bmod m$ értékét kapjuk, amit megszorzunk r inverzével, hogy megkapjuk a-t. Ennek hatására a művelet nem csak a kriptoszövegtől fog függeni, és a dekódolás ideje is véletlenszerűen fog ingadozni. Hasonló elven működik az *elágazás-előrejelzős támadás* (branch prediction attack), amikor is a modern processzorok elágazás-előrejelző egységének működéséből következtetünk arra, hogy bizonyos elágazásokon merre haladt tovább a program, és ebből visszakövetkeztetünk a kulcs bitjeire.

Meg kell még jegyeznünk emellett, hogy az sem bizonyított, hogy az $a^x \bmod m$ függvény inverzét kizárólag az m prímtényezős felbontásának ismeretében lehet kiszámolni. Ha bárki talál erre egy hatékonyabb módot, az összes RSA implementációt képes lesz támadni vele, függetlenül attól, hogy mennyire körültekintően programozták azokat.[2]

4. Az implementáció részletei

4.1. Az implementálandó algoritmusok

A legfelső szinten az RSA algoritmushoz két művelet szükséges: adott hosszúságú prímszám generálása a kulcsgeneráláshoz, és moduláris hatványozás a titkosításhoz és a visszafejtéshez. A prímszám generálása úgy történik, hogy egy véletlenszámból indulunk ki, és prímteszteket futtatunk az utána következő számokra. A prímtesztelés többlépcsős folyamat, ami egy triviális teszttel kezdődik (oszthatóság 2-vel, 3-mal és 5-tel), majd próbaosztásokat végzünk kis prímekkel, végül egy Miller–Rabin tesztet hajtunk végre. A Miller–Rabin teszt moduláris hatványozást alkalmaz, így újra felhasználhatjuk a titkosítás kódját. A moduláris hatványozáshoz szükségünk van hosszú számok szorzására, a szorzás speciális optimalizált eseteként a négyzetre emelésre, valamint maradékos osztásra. Mint látni fogjuk, a maradékos osztás helyettesíthető a hatékonyabb Montgomery-redukcióval. Az egyéb szükséges segédalgoritmusok igénylik hosszú számok összeadását, kivonását és osztását is, de ezeket ritkán használjuk, így nem szükséges őket sebességre optimalizálni.

[2]Léteznek olyan titkosítások is, melyek feltörése bizonyítottan olyan nehéz, mint a faktorizáció, bár az RSA-nál kevésbé elterjedtek.

4.2. A felhasznált adatszerkezetek

A hosszú számokat a gépi szóval megegyező méretű „szeletekben” érdemes tárolni, hiszen ez az a méret, amellyel az architektúra a leghatékonyabban képes dolgozni. Ezeket a „szeleteket” lista adatszerkezetben kell tárolnunk, hiszen jól definiált sorrendjük van. Innentől két választásunk van: tárolhatjuk a „szeleteket” helyi érték szerint növekvő vagy csökkenő sorrendben.[3] Én a növekvő sorrendet választottam, két okból:

1. A legtöbb felhasznált algoritmus az alacsonyabb helyi értékektől a magasabbak felé halad, a gyorsítótárazási (cache) technikák pedig általában azt feltételezik, hogy az algoritmusok a memóriát címek szerint növekvő sorrendben írják és olvassák. A helyi érték szerint növekvő tárolás esetén tehát abban a sorrendben olvassuk a memóriát, ahogy a cache „elvárja” tőlünk.

2. Az x86-os architektúra maga is ezt a sorrendet alkalmazza, amikor a gépi szavait a memóriába írja: a legkisebb helyi értékű bájt kerül a legkisebb címre. Ha szükséges, a számot tekinthetjük helyi érték szerint növekvő bájtok sorozatának is, függetlenül a gépi szó hosszától.

A lista adatszerkezet implementációjához a C++ standard könyvtárában található `std::vector` osztályt használtam. Ez a memóriában folytonosan tárolja az elemeket, dinamikusan újrafoglalva a területet, ha a régi már nem elegendő az elemek tárolására. Felhasználása nagyon hasonlít a hagyományos tömbökére, kivéve, hogy a mérete módosítható futás közben. Lehetőséget nyújt az indexelés ellenőrzésére is, de ezt csak hibakeresési célból érdemes bekapcsolni, mert csökkenti a teljesítményt. A folytonos tárolás lehetővé teszi, hogy assembly kódból hagyományos tömbként kezeljük a tartalmát, de ekkor nem módosíthatjuk a méretet, és elvesztjük az indexellenőrzés lehetőségét is. Kihasználva a dinamikus méretezhetőség lehetőségét, a program mindig eltávolítja a számokból a vezető nullákat: a legutolsó elem sohasem nulla (az alacsony szintű rutinokon kívül). Ennek a szabálynak megfelelve a nulla értéket egy üres vektor jelzi.

Eddig csak a nemnegatív számok tárolásáról esett szó. Az előjeles számok tárolására csak a multiplikatív inverz számításakor van szükség; ehhez egy saját struktúrát használok, amely külön tárolja az előjelet és a szám értékét (negatív előjel esetén kettes komplemens alakban). A kettes komplemens alak lehetővé teszi, hogy az összeadást és kivonást az előjeltől független algoritmus végezhesse, és a műveletből kijövő carry alapján tudjuk korrigálni az előjelet, ha szükséges. A fenti, vezető nullákra vonatkozó szabály

[3] Persze elméletileg nem csak ez a két lehetőségünk van – a „szeleteket” tárolhatnánk ezek tetszőleges permutációjaként is. Ezzel viszont csak feleslegesen növelnénk az algoritmusaink bonyolultságát, így a gyakorlatban csak a fenti két lehetőség merül fel.

úgy módosul, hogy pozitív szám esetén nulla, negatív szám esetén csupa egyes bitből álló szám nem lehet az utolsó elem.

4.3. Az architektúra részletei

Az x86 architektúra modern processzorai csővezetékes és szuperskalár felépítésűek. A csővezetékes feldolgozás azt jelenti, hogy a processzor egyszerre több utasításon is dolgozik, amelyek a feldolgozás különböző fázisaiban vannak (dekódolás, operandusok betöltése, végrehajtás, stb.). A szuperskalár felépítés esetén bizonyos feldolgozó részegységekből többet is tartalmaz a processzor, és ezek egymással párhuzamosan képesek dolgozni; a legújabb AMD processzorokban például három aritmetikai/logikai egység (ALU) is található. Az x86 komplex utasításkészletű (CISC) architektúra, ami azt jelenti, azaz egyetlen utasítás több elemi lépést is végezhet (pl. betöltés memóriából, számítás, majd visszaírás a memóriába); ezért a modern processzorokba bonyolult dekódoló áramköröket kell építeni, melyek képesek lebontani az utasításokat elemi lépésekre, ún. mikroműveletekre (micro-op). Az architektúra eredeti tervezésekor nem volt követelmény a párhuzamosítás, ezért a gépi kód nem is tartalmaz információt arról, hogy mely műveletek végezhetők párhuzamosan; a szuperskalár processzoroknak végrehajtás közben kell feltérképezniük az utasítások közötti függőségeket, és késleltetni azon műveleteket, amelyek bemenő paraméterei még nem állnak készen. Mindezeket összevetve, bár tetszőleges régi x86-os gépi kód lefut a legújabb processzoron is, a hatékony kód írásához teljesen más hozzáállásra van szükség egy 386-os processzornál, mint egy AMD Opteronnál vagy egy Intel Core 2-nél.

Az általános célú regiszterek száma 32 bites módban 8, 64 bites módban 16; ezekből az egyik a veremmutató, amit csak nagyon speciális esetben szokás másra használni. A 8 regiszter nagyon kevésnek számít, és általában megakadályozza, hogy az összes felhasznált részeredményünket regiszterben tartsuk. A 16 regiszter, bár sokkal jobb, még mindig kevésnek számít a redukált utasításkészletű (RISC) processzorok 32-64 regiszteréhez képest.

Az x86-os architektúrák a következő alapvető aritmetikai műveleteket biztosítják számunkra:

ADD — Két gépi szó összeadása, és az eredmény tárolása az egyik operandus helyén. Az esetlegesen fellépő túlcsordulást egy speciális bit (carry flag) beállításával jelzi. Egyik változata (ADC) képes arra, hogy az eredményhez a carry flag értékét is hozzáadja, ezt közvetlenül felhasználhatjuk hosszú számok összeadásakor

SUB — Két gépi szó kivonása, és az eredmény tárolása a kisebbítendő helyén. Az alulcsordulást a carry flag jelzi. Egyik változata (SBB) a carry flag értékét

is kivonja a kisebbítendőből, ezt közvetlenül felhasználhatjuk hosszú számok kivonásakor.

MUL Két gépi szó szorzása, és az eredmény tárolása két rögzített regiszterben. Az eredmény legfeljebb akkora lehet, mint az operandusok hosszának összege, ezért túlcsordulás nem fordulhat elő. Létezik olyan variánsa is (IMUL), amely csak az eredmény alsó 32 bitjét adja vissza.

DIV Két gépi szó hosszú szám egészosztása egy gépi szóval, az eredmény és a maradék tárolása egy-egy rögzített regiszterben. Ha az eredmény nem fér el egy gépi szón, speciális hibajelzés (kivétel) keletkezik, és az eredményt egyáltalán nem kapjuk meg. Az utasítás a többiekhez képest nagyon lassú, ezért használatát érdemes minimalizálni.

SHL/SHR Bitenkénti balra/jobbra tolás, alkalmazható a 2 hatványaival való gyors szorzásra/osztásra. A jobbra tolásnak van olyan verziója (SAR), amely megtartja az operandus előjelét, így előjeles számok osztására is alkalmas. Mindegyik változat megadja a legutoljára „kitolt" bitet a carry flagben.

A fentieken kívül természetesen rendelkezésre áll mozgató utasítás (MOV) a regiszterek közti mozgatáshoz és a memória írásához/olvasásához, valamint feltételes és feltétel nélküli ugró utasítások (JMP, Jxx) az elágazások és ciklusok megvalósításához. Látható, hogy bár a műveletek hasonlítanak a C++ aritmetikai műveleteihez, nem tökéletes a fedés a kettő között: a C++ szorzás eredménye például mindig olyan hosszú, mint az operandusok, az összeadásnál pedig nem tudjuk detektálni a túlcsordulást. Ez a tény megakadályoz minket abban, hogy a lehető legnagyobb aritmetikai teljesítményt érjük el tiszta C++ nyelv használatával, a fordítónk képességeitől függetlenül.

Meg kell még említenünk, hogy a modern x86 processzorok képesek SIMD (Single Instruction, Multiple Data) utasítások végrehajtására is; ezek képesek a regiszterek egyes „szeleteit" önálló adatként értelmezve egyszerre több adaton elvégezni ugyanazt a műveletet (pl. a 64 bites regiszter tartalmát 4 db. 16 bites számként kezelve összeadni egy másik regiszter 4 db. 16 bites számával). Ez nagyon hasznos pl. multimédia kezeléséhez, ahol ugyanazt a műveletsort kell végrehajtani egymástól független képpontokra. A mi célunkból sajnos nem olyan hasznosak ezek az utasítások, mert a mi algoritmusaink során általában keletkezik valamilyen átvitel, amit fel kell használni a következő lépésben, ez pedig megnehezíti a párhuzamos végrehajtást. Ettől függetlenül, mint a szorzásnál látni fogjuk, egyes architektúrákon még akkor is kifizetődő az SIMD utasítással végzett szorzás, ha egyszerre csak egy szorzást végzünk vele.

5. A felhasznált algoritmusok

A program által felhasznált algoritmusokat két csoportra lehet osztani:

- A „magas szintű” algoritmusok a számokat egy egységként kezelik, ezért függetlenek a számok hosszától. Ezeket rövid számokra is ugyanígy lehetne implementálni, mint hosszú számokra; a különbség csak annyi, hogy míg rövid számokon a számítógép közvetlenül is el tudja végezni az alapműveleteket, hosszúszámokon segédalgoritmusokra (az „alacsony szintű” algoritmusokra) van szükség. Optimalizálásuk általában abban merül ki, hogy minimálisra csökkentjük az elvégzett alapműveletek számát; az assembly nyelvű megvalósítás nem hozna jelentős teljesítménynövekedést.

- Az „alacsony szintű” algoritmusok egyszerre egy gépi szót dolgoznak fel, ilyen műveletekkel valósítják meg a hosszúszámok alapműveleteit. Teljesítményük erősen függ attól, hogy az architektúránk milyen műveleteket képes elvégezni, és hogy ezeket mennyire hatékonyan használjuk ki. Ha maximális teljesítményre van szükségünk, kifizetődő lehet ezeket az algoritmusokat assembly kóddal implementálni.

5.1. Magas szintű algoritmusok

5.1.1. A moduláris hatványozás

Mint láttuk, az RSA titkosítás hatékony implementációja gyakorlatilag egyet jelent az $a^x \bmod m$ érték hatékony kiszámításával. Ezt nem bonthatjuk szét hatványozásra és maradékképzésre – a és x legalább 512 bit hosszúak, ezért a hatvány elfogadhatatlanul sok helyet foglalna a memóriában csak azért, hogy a végső maradékképzésnél egy rövid számot kapjunk belőle. Ehelyett a maradékképzést minden szorzás után el kell végeznünk, kihasználva, hogy $(ab) \bmod m = ((a \bmod m)b \bmod m$.

A hatványozást nem végezhetjük ismételt szorzással, hiszen 2^{512} darab szorzás kivárhatatlanul sok ideig tartana. Hogy ezt elkerüljük, vegyük észre, hogy $a^{2x} = (a^x)^2$, valamint $a^{2x+1} = a(a^x)^2$; másképpen mondva, egy l bites kitevőjű hatvány kiszámítása visszavezethető egy $l-1$ bites kitevőjű hatvány kiszámítására, egy négyzetre emelésre, és esetleg egy szorzásra. Ez a rekurzió nyilván nem végtelen, $x = 0$ elérésekor megszakad ($a^0 = 1$), így a hatvány kiszámításához végül l darab négyzetre emelés és $\lambda(a)$ darab szorzás szükséges. ($\lambda(a)$ az a 1-es bitjeinek számát jelöli.) Ezt az algoritmust „balról jobbra” hatványozásnak nevezzük (algoritmus 1), mivel a kitevő bitjeit balról jobbra haladva használjuk fel.[4] A szorzások után természetesen beilleszthetünk egy maradékképzést, így hatványozó

[4]Létezik „jobbról balra” hatványozási algoritmus is, amely ugyanannyi idő alatt fut, mint a „balról jobbra” algoritmus, de mint látni fogjuk, a mi esetünkben a „balról jobbra” algoritmus továbbfejleszthető. A „jobbról balra” algoritmus akkor fejleszthető tovább, ha többször használjuk ugyanazzal az alappal, pl. az ElGamal titkosító algoritmusnál.

Algoritmus 1 „Balról jobbra” hatványozás

```
hatványoz(alap, kitevõ):
    eredmény := 1
    for i:=hossz(kitevõ)-1 downto 0:
        eredmény := eredmény*eredmény
        if (a kitevõ i. bitje 1):
            eredmény:= eredmény*alap
    return eredmény
```

Algoritmus 2 „t-szeres” hatványozás

```
hatványoz(alap, kitevõ, t):
    segéd[0]:=1
    for i:=1 to (2^t)-1 do:
        segéd[i]:=segéd[i-1]*alap
    eredmény:=1
    for i:=(hossz(kitevõ)/t)+1 downto 0 do:
        for j:=1 to t do:
            eredmény:=eredmény*eredmény
        számjegy:=(a kitevõ (i*t)..((i+1)*t-1) bitjei)
        eredmény:=eredmeny*seged[szamjegy]
    return eredmény
```

algoritmus helyett modulárisan hatványozó algoritmust kapunk.

Most vegyük észre, hogy a fenti elvet tovább lehet gondolni más szorzókkal is; általánosan úgy írhatjuk fel, hogy $a^{nx+k} = a^k(a^x)^n$. Ekkor az n-edik hatványozások száma $\log_n x$ lesz, a szorzások száma pedig annyi, amennyi nullától különböző számjegy van x n-es számrendszerbeli alakjában (feltéve, hogy a^2, $a^3 \ldots a^{k-1}$értékeit előre kiszámítjuk). Mivel bináris számítógépen dolgozunk, érdemes az n-et 2^t alakúnak választani, hogy a számjegyek egyszerű bitműveletekkel kinyerhetőek legyenek a bináris alakból. Ekkor az n-edik hatványra emelést megoldhatjuk t darab négyzetre emeléssel. Végeredményben továbbra is l darab négyzetre emelést fogunk végezni, de a szorzások száma már csak $\frac{l}{t}$ lesz; ez a „t-szeres” hatványozás (algoritmus 2).

Nézzük meg kicsit az a hatványait tartalmazó segédtáblázatot! Megfigyelhetjük, hogy a páros kitevőjű hatványok igazából feleslegesek – pl. $a^6 = (a^3)^2$, $a^{20} = ((a^5)^2)^2$ és így tovább. Ezeket a négyzetre emeléseket egyébként is el kell végeznünk, ezért mindegy, ha a szorzás után végezzük őket, ezzel megspórolva a páros kitevőjű hatványok tárolását. A kitevőben továbbra is balról jobbra haladunk, és amíg nullás bitet találunk, csak négyzetre emelést végzünk. Ha egyes bithez értünk, megkeressük a leghosszabb olyan bitsorozatot, amely a t-nél még nem hosszabb és egyesre végződik. Legyen ennek a sorozatnak az értéke

Algoritmus 3 „csúszó ablak” hatványozás

```
hatványoz(alap, kitevõ, t):
    segéd[0]:=alap
    négyzet:=alap*alap
    for i:=1 to (2^(t-1))-1 do:
        segéd[i]:=segéd[i-1]*négyzet
    eredmény:=alap
    p:=hossz(kitevõ)-2
    loop:
        while (a kitevõ p-edik bitje 0) and (p>0):
            eredmény:=eredmény*eredmény
            p:=p-1
        if p=0:
            break
        s:=(a p. bittõl kezdõdõ leghosszabb egyesre végzõdõ
             bitsorozat, legfeljebb t bit)
        l:=hossz(s)
        for i:=1 to l do
            eredmény:=eredmény*eredmény
        eredmény:=eredmény*segéd[(s-1)/2]
    return eredmény
```

s, a hossza l. Ekkor el kell végeznünk még l darab négyzetre emelést, hogy a részeredményünk kitevőjének végén l darab nullás bit legyen, azután szorozhatunk a^s-nel. Ezután visszaléphetünk a négyzetre emelő fázisba. Ezt a módszert „csúszó ablak” algoritmusnak nevezik (algoritmus 3); úgy képzelhetjük el, hogy egy t bit széles ablak csúszik végig a kitevő bitjein, amíg a lehető legtöbb egyes bitet „be nem fogja”. A négyzetre emelések száma továbbra sem változott. A szorzások számának meghatározása jóval bonyolultabb lett, mint az előző két esetben – pontos kiszámítására nem is vállalkoznék, de talán ránézésre is látszik, hogy az előző két algoritmusnál általában kevesebb. A programom ezt az algoritmust használja a moduláris hatványozáshoz.

Látható, hogy a leggyakrabban végzett művelet a maradékképzés (minden szorzás és négyzetre emelés után), a második leggyakoribb a négyzetre emelés, a maradék idő nagy részét pedig szorzások teszik ki. Mint később látni fogjuk, a maradékképzés helyettesíthető egy hatékonyabb, Montgomery-redukció nevű művelettel páratlan osztó esetén. Szerencsére az osztónk vagy egy páratlan prím, vagy két páratlan prím szorzata, így nem akadályoz minket semmi abban, hogy ezt a műveletet használjuk.

5.1.2. Prímgenerálás

Mint azt a 4.1 szakaszban már említettem, a prímszám generálása egy megfelelő méretű véletlenszámból indul ki, és három fázisból, vagy ha úgy tetszik, „szűrőből" áll. Ha egy szám „fennakad" valamelyik szűrőn, a többit már nem alkalmazzuk rá. A fázisok a következők:

Kerékmódszer. Első körben kiszűrjük azon számokat, amelyek kettővel, hárommal vagy öttel oszthatóak. Ehhez az ún. *kerékmódszer*t használjuk. Megadunk egy listát azon 30 alatti számokkal, amelyek nem oszthatók a fenti három prímmel. (8 darab ilyen szám van: 1, 7, 11, 13, 17, 19, 23, 29) Tudjuk, hogy ha egy szám 30-cal vett maradéka ezen számok valamelyike, akkor maga a szám sem osztható ezekkel a prímekkel. Kiindulunk tehát egy 30-cal osztható véletlenszámból, és ehhez hozzáadjuk egyenként a lista elemeit – ezek lesznek a következő fázisba továbbhaladó számok. Ha ezek közül egyik sem jut át a maradék két fázison, akkor 30-cal növeljük a kiinduló értéket, és újra hozzáadjuk a lista elemeit. Egy ilyen szakaszra mondjuk azt, hogy „megforgattuk a 30-as kereket". Ez a fázis a számok $\frac{8}{30} \simeq 0.2667$ részét engedi át.[5]

Próbaosztás. A kerékmódszeren átjutott számokat megpróbáljuk elosztani a 7 és 65521 közötti összes prímszámmal. Összesen 6539 darab ilyen prím van, de általában nincs szükség arra, hogy mindegyiket végigpróbáljuk – a legtöbb szám már a 7-nél vagy a 11-nél „megbukik". Magának a próbaosztásnak az implementációja egy alacsony szintű algoritmus, ezért itt nem tárgyaljuk.

Miller–Rabin prímteszt. Az utolsó fázis egy probabilisztikus prímteszt. Ez azt jelenti, hogy a teszthez véletlenszerű bemenetet használunk, és az összetett számok csak egy bizonyos valószínűséggel fognak megbukni a teszt során. A tesztet egymás után többször elvégezve egyre jobban növelhetjük annak a valószínűségét, hogy a szám prím, de sohasem érhetjük el a teljes bizonyosságot. Ezért is szokás az így generált prímeket „ipari minőségű" prímeknek nevezni. A gyakorlatban nem probléma a bizonyosság hiánya – az összetett szám „átengedésének" valószínűsége olyan csekély, hogy nyugodtan elhanyagolhatjuk.

Az algoritmus alapelve a kis Fermat-tételből indul ki. Ha p prím, és $1 < a < p$, akkor $a^{p-1} \equiv 1 \pmod{p}$, vagy másképpen $a^{p-1} - 1 \equiv 0 \pmod{p}$. Tudjuk továbbá, hogy p páratlan, vagyis $p - 1$ felírható $2^k d$ alakban, azaz

$$a^{2^k d} - 1 \equiv 0 \pmod{p}$$

[5] A módszert nyilván tovább lehet vinni több prímszámra is, de a relatív haszna egyre csökken: a segédtáblázat mérete a kiszűrendő prímek szorzatával arányos, a kiszűrt számok aránya viszont egyre lassabban nő.

Ekkor felhasználhatjuk az $a^2 - b^2 = (a+b)(a-b)$ azonosságot:

$$\left(a^{2^{k-1}d} + 1\right)\left(a^{2^{k-1}d} - 1\right) \equiv 0 \pmod{p}$$

A második tagra újra felhasználhatjuk az azonosságot, és ezt addig ismételhetjük, amíg van kétszeres szorzó a kitevőben. Végeredményként ezt kapjuk:

$$\left(a^{2^{k-1}d} + 1\right)\left(a^{2^{k-2}d} + 1\right)\ldots\left(a^{2d} + 1\right)\left(a^{d} + 1\right)\left(a^{d} - 1\right) \equiv 0 \pmod{p}$$

Ez a szorzat csak akkor lehet nullával kongruens, ha legalább az egyik tagja az, feltéve, hogy p prím. Ezt átfogalmazva, ha p prím, akkor az alábbi két állítás legalább egyike igaz:

$$a^d \equiv -1 \pmod{p}$$

$$a^{2^i d} \equiv 1 \pmod{p}, \text{ahol } 0 \leq i < k$$

Ennek ismeretében az algoritmus a következő:

1. Legyen a tesztelendő szám p! Válasszunk egy $1 < a < p$ véletlenszámot! Ha a osztja p-t, akkor p nyilvánvalóan összetett, így végeztünk.

2. Számoljuk meg a $p-1$ végén levő nullás biteket, ez lesz k. Ezeket a biteket levágva kapjuk d-t.

3. Számítsuk ki a^d mod p értékét egy moduláris hatványozó algoritmussal! Ha az eredmény 1 vagy $p-1$, akkor az összetettség nem bizonyítható, és kilépünk.

4. Az előző lépésben kapott értéket emeljük négyzetre modulárisan $k-1$-szer! Ha bármelyik lépés után $p-1$ az eredmény, akkor az összetettség nem bizonyítható. Ha egyik lépés után sem kapunk $p-1$-et, akkor a p nem lehet prím, a fenti tétel miatt.

Bizonyítható, hogy egy adott összetett p esetén a lehetséges a számok legalább $\frac{3}{4}$-e bizonyítja az összetettséget. t darab teszt elvégzése után tehát annak a valószínűsége, hogy p összetett, legfeljebb 4^{-t}. A program 30 tesztet végez, mielőtt elfogadná a számot prímként, vagyis kevesebb, mint egy a trillióhoz az esélye, hogy összetett számot kapunk végeredményül. Mint látszik, ez a teszt önmagában is elég megbízhatóan eldönti, prímszámmal dolgozunk-e, viszont a moduláris hatványozás miatt sokkal lassabb, mint a fenti két teszt. Sokkal jobban megéri a „könnyű” eseteket ezekkel kiszűrni, mint vakon minden számra alkalmazni a Miller–Rabin tesztet.

5.1.3. A kínai maradéktétel alkalmazása

Ha az RSA dekódolás bonyolultabb, párhuzamosítható változatát akarjuk megvalósítani, szükségünk van olyan módszerre, amely $x_1 = a \bmod p$ és $x_2 = a \bmod q$ értékéből visszaadja $x = a \bmod pq$ értékét. A kínai maradéktétel bizonyítását használva olyan algoritmust kapunk, amelynek szüksége van $p^{\varphi(q)} \bmod q$ és $q^{\varphi(p)} \bmod p$ értékekre, és ezekből két moduláris szorzással és egy moduláris összeadással kapja a végeredményt [Knuth, 286. o.]. Ezen javíthatunk, ha H. L. Gardner algoritmusát használjuk [Knuth, 290. o.][Handbook, 612. o.]: ekkor előre ki kell számítanunk $p^{-1} \bmod q$ értékét. Gardner algoritmusa tetszőleges számú maradékra alkalmazható, és mindig hatékonyabb a hagyományos módszernél. Két maradék esetén a következő számításra egyszerűsödik:

$$x = x_1 + ((x_2 - x_1)p^{-1} \bmod q)p$$

Az eredmény p-vel vett maradéka x_1, mivel a második tag p többszöröse. A q-val vett maradék:

$$x_1 + (x_2 - x_1)p^{-1}p = x_1 + x_2 - x_1 = x_2$$

Látjuk tehát, hogy a kapott x érték 0 és pq között van, p-vel vett maradéka x_1, q-val vett maradéka x_2, vagyis teljesíti a követelményeket. Kiszámításához egy moduláris kivonásra, egy moduláris szorzásra, egy hagyományos szorzásra és egy összeadásra van szükség. A moduláris összeadás és kivonás megoldható maradékos osztás nélkül, míg a szorzás nem, így végeredményben eggyel kevesebb maradékos osztást kell végeznünk, mint a hagyományos esetben.

5.1.4. A moduláris inverz kiszámítása

A moduláris inverz számítására szükség van a fenti algoritmuson kívül a titkos kulcs $d_1$és d_2 dekódoló kitevőinek számításakor is. Mint az algoritmus leírásánál is említettem, a kibővített euklideszi algoritmus alkalmas a feladatra. Ez az algoritmus sajnos nem igazán hatékony hosszú számok esetén, mivel maradékos osztást használ, amelynek futásideje a bemenet négyzetével arányos. Sokkal hatékonyabb lenne egy olyan módszer, amely csak lineáris időben futó műveleteket használ fel.

Szerencsére létezik ilyen algoritmus, a neve „bináris legnagyobb közös osztó” [Knuth, 338. o.]. Azt használja ki, hogy bináris számítógépeken a kettő hatványaival való osztás lineáris időben, bitenkénti eltolással végezhető. A következő egyszerű tényeket használjuk ki:

1. Ha x és y párosak, akkor $\mathrm{lnko}(x, y) = 2\mathrm{lnko}(x/2, y/2)$
2. Ha x páros és y páratlan, akkor $\mathrm{lnko}(x, y) = \mathrm{lnko}(x/2, y)$

3. Tetszőleges x és y egészekre $\text{lnko}(x, y) = \text{lnko}(x - y, y)$

4. Ha x és y is páratlan, akkor $|x - y|$ páros, és $|x - y| < \max(x, y)$

Ezekből az állításokból mindig felhasználhatunk egyet arra, hogy x és y közül legalább az egyiket csökkentsük. Érdemes először az 1. pontot felhasználva eliminálni az x és y közös kettes szorzóit, majd a megmaradó x' és y' számokra alkalmazni a maradék három pontot. Ebben a második fázisban az lnko mindig állandó marad, miközben a felhasznált számok folyamatosan csökkennek; x és y sohasem lehet egyszerre páros ebben a fázisban. Amikor a kisebbik 0 lesz, a nagyobbik fogja tartalmazni $\text{lnko}(x', y')$-t. Ezt még meg kell szoroznunk az első fázisban kiemelt kettes szorzókkal, hogy megkapjuk a végső legnagyobb közös osztót. A Miller–Rabin teszt során ezt az algoritmust alkalmaztam, annyi módosítással, hogy egyből kilépünk, ha az lnko értéke biztosan egynél nagyobb; az osztó pontos értéke nem számít az algoritmus során.

Ez az algoritmus önmagában nem elegendő a moduláris inverz kiszámítására. Tovább kell fejlesztenünk a módszert: a kiterjesztett euklideszi algoritmus mintájára „kiterjesztett bináris lnko"-nak nevezzük az így kapott algoritmust [Handbook, 608. o.]. Célunk olyan a és b számok meghatározása, amelyekre igaz, hogy $ax + by = \text{lnko}(x, y)$. Az első fázist változatlanul hagyhatjuk, mivel x, y és az lnko feleződnek, változatlan a és b mellett is igaz marad az egyenlet. A második fázisban fenntartunk A, B, C és D segédértékeket úgy, hogy mindig igazak legyenek a $Ax' + By' = u$ és $Cx' + Dy' = v$ egyenletek. A kezdőértékek $A = 1$, $B = 0$, $C = 0$, $D = 0$, $u = x'$, $v = y'$, ezekre igazak az egyenletek. A fenti pontokat a következőképpen egészíthetjük ki:

1. Ha u páros és v páratlan, akkor $u' = u/2$, A és B értékére pedig két eset lehetséges

   (a) Ha A és B mindketten párosak, akkor $A' = \frac{A}{2}$ és $B' = \frac{B}{2}$.
   $A'x' + B'y' = \frac{A}{2}x' + \frac{B}{2}y' = \frac{Ax'+By'}{2} = \frac{u}{2} = u'$, vagyis az egyenlőség továbbra is fennáll.

   (b) Ha A vagy B páratlan, akkor $A' = \frac{A+y'}{2}$ és $B' = \frac{B-x'}{2}$.
   $A'x' + B'y' = \frac{A+y'}{2}x' + \frac{B-x'}{2}y' = \frac{Ax'+x'y'+By'-x'y'}{2} = \frac{Ax'+By'}{2} = \frac{u}{2} = u'$, az egyenlőség itt is fennáll. Az esetek végigpróbálásával az is belátható, hogy A és y', valamint B és x' párosságának egyeznie kell, így A' és B' is egész számok lesznek.

2. Ha v páros és u páratlan, teljesen analóg a gondolatmenet v-vel, C-vel és D-vel

3. Ha u és v mindketten páratlanok, akkor:

   (a) Ha u a nagyobb, $u' = u - v$, $A' = A - C$, $B' = B - D$.
   $A'x' + B'y' = (A - C)x' + (B - D)y' = (Ax' + By') - (Cx' + Dy') = u - v = u'$.

(b) Ha v a nagyobb, $v' = v - u$, és a gondolatmenet ugyanaz.

Mint látjuk, u és v közül legalább az egyik minden lépésben kisebb lesz, amíg egyikük el nem éri a nullát. A 3. pont gondoskodik róla, hogy mindig az u legyen a kisebb, így az fog hamarabb nullázódni. Mikor ezt elértük, $v = \text{lnko}(x', y')$, és továbbra is igaz, hogy $Cx' + Dy' = v$, vagyis $a = C$ és $b = D$ végeredménnyel sikeresen elvégeztük az algoritmust. Sikerült elkerülni az osztásokat, cserébe több menetet kell végezni összeadásokkal, kivonásokkal és bitenkénti jobbra tolásokkal. Megfigyelhetjük, hogy u és v sohasem lesz negatív, az A, B, C, D segédértékek viszont lehetnek, ezért ezeket előjelesen kell ábrázolni.

Ha az $x^{-1} \bmod m$ moduláris inverzre vagyunk kíváncsiak, akkor a kiterjesztett bináris lnko algoritmust kell futtatnunk x és m értékeire. Moduláris inverz csak akkor létezhet, ha x és m relatív prímek, így a közös kettes szorzók kiemelését el is hagyhatjuk az algoritmusból. A kapott $ax + bm = 1$ végeredményt átírhatjuk $ax \equiv 1 \pmod{m}$ alakba, ahonnan már látszik, hogy $a \equiv x^{-1} \pmod{m}$.[6]

5.2. Alacsony szintű algoritmusok

Mint fentebb említettem, ezek az algoritmusok egyszerre egy gépi szón dolgozva valósítják meg a hosszúszámok alapműveleteit. Ezen algoritmusok tárgyalásakor érdemes úgy venni, hogy a feldolgozott számok $B = 2^b$ alapú számrendszerben vannak felírva, ahol b a számítógépünk gépi szavának hossza. Ha így nézzük, a számítógép és az emberek számolási képességei hasonlítanak egymáshoz: az emberek fejben össze tudnak adni két számjegyet, a számítógépek két gépi szót; az emberek fejben össze tudnak szorozni két számjegyet (ezért tanuljuk meg a szorzótáblát), a számítógépek össze tudnak szorozni két gépi szót; az osztásnál is hasonlóak a gyengeségeink, ezért sokkal bonyolultabb algoritmusra van szükségünk, mint szorzásnál. Épp ezért egy-egy algoritmus megértéséhez hasznos lehet ugyanazt papíron, tízes számrendszerben végigszámolni, amit majd a számítógép gépi szavakkal fog végezni.

5.2.1. Összeadás, kivonás

Ezen műveletek teljességgel az általános iskolában tanult írásbeli összeadás és kivonás analógiájára végezhetők, futási idejük a bemenet lineáris függvénye. A futási időt általában a négyzetes idejű vagy lassabb algoritmusok dominálják, ezért ezt a két műveletet nem szükséges optimalizálni.

[6] Lehet, hogy az a értékét negatív számként kapjuk meg. Ekkor érdemes hozzáadni m-et, hogy a továbbiakban szokásos előjel nélküli számként kezelhessük.

5.2.2. Szorzás

Ez a művelet is teljesen analóg az általános iskolás írásbeli szorzással, mégis jóval több figyelmet érdemel, mint az összeadás és a kivonás. Ennek egyik oka, hogy négyzetes időben fut, ezért sokkal fontosabb az optimalizációja. A másik fontos ok, hogy több, a szorzáshoz többé-kevésbé hasonló alacsony szintű műveletet fogunk tárgyalni, és ezek megértéséhez előny a hagyományos szorzás megértése.

Amikor papíron összeszorzunk egy k számjegyű számot egy l számjegyűvel, tulajdonképpen kl darab egyszerű szorzást végzünk, a helyi értékeket pedig összeadjuk:

$$X \times Y = (x_0 B^0 + x_1 B^1 + \ldots + x_{k-1} B^{k-1})(y_0 B^0 + y_1 B^1 + \ldots + y_{l-1} B^{l-1}) = \sum_{i=0}^{k-1} \sum_{j=0}^{l-1} x_i y_j B^{i+j}$$

A B-s tagokat nem kell kiszámolnunk, hanem a végeredményben elfoglalt pozíciót jelölik ki. $x_i y_j$ nem lehet hosszabb két számjegynél, ugyanis x_i és y_i legnagyobb értéke $B-1$ lehet, a maximális szorzat pedig $(B-1)^2 = B^2 - 2B + 1$. Az egyes tagok összeadásának sorrendje matematikai szempontból lényegtelen, de implementációs szempontból annál fontosabb – a cache kihasználása érdekében az az előnyös, ha a memóriát folytonosan, növekvő címsorrendben érjük el.

A tagokon két egymásba ágyazott ciklussal haladunk végig: a külső az első szám jegyein halad végig növekvő helyi érték szerint, a belső a másodikén, elvégezve a szorzásokat, és hozzáadva a kapott értéket az eddigi részeredményhez.[7] Az egyes szorzások eredményét nem adjuk egyből hozzá a memóriában található célváltozóhoz, mert akkor minden memóriacímet kétszer kellene írni a belső ciklusban: egyszer az $x_i y_{j-1}$ felső számjegyét kellene hozzáadni, aztán az $x_i y_j$ alsó számjegyét. Ehelyett a szorzat felső számjegyét egy regiszterben tároljuk, amit a következő lépésben hozzáadunk az ott megkapott szorzathoz. Az előző szorzatból áthozott számjegy legfeljebb $B-1$ értékű lehet, vagyis az összeadás után legfeljebb $B^2 - 2B + 1 - (B-1) = B^2 - B$ lehet a kapott érték. Ehhez hozzá kell adnunk a célcímen már meglevő, legfeljebb $B-1$ értékű számjegyet, így a maximális érték $B^2 - 1$ lesz, ami még éppen hogy elfér két számjegyen. A két összeadás után az eredmény alsó számjegyét visszaírjuk a célcímre, a felső pedig továbbmegy a következő iterációba.

Tanulságos megnézni a belső ciklus x86 assembly nyelven írt megvalósítását:

```
1 ; az ESI az aktuális forráscímet, az EDI a célcímet tartalmazza
2 ; az EBX a külsõ ciklus aktuális számjegye, ezzel szorzunk mindent
3 ; az EBP tartalmazza az elõzõ iterációból maradt carry értéket
```

[7] Igazából ez egy szorzó-összeadó algoritmus, mivel nem kötöttük ki, hogy a célként megjelölt memóriaterület nulla értékről indul. Ez hasznos lehet akkor, ha egy algoritmus a szorzás után közvetlenül összeadást is igényel, mert az összeadási lépést nem kell külön végrehajtani. Ha csak szorzásra van szükségünk, a célterületet ki kell nullázni a külső ciklus indítása előtt.

```
4  .inner_loop:
5      mov eax,[esi]     ; forrás számjegy betöltése az EAX regiszterbe
6      mul ebx           ; az EAX szorzása az EBX-szel, az eredmény az EDX:EAX
7                        ; regiszterpárba kerül
8
9      add eax,ebp       ; a carry hozzáadása az EAX-hez
10     adc edx,0         ; az elõzõ összeadás carry-jének átvitele az EDX-be
11                       ; (64 bites összeadás)
12
13     add [edi],eax     ; EAX hozzáadása a célcímhez
14     mov ebp,edx       ; a carry átmozgatása az EBP-be
15     adc ebp,0         ; az elõzõ összeadás carry-jének átvitele az EBP-be
16                       ; (64 bites összeadás)
17
18     add esi,4         ; forrás- és célcím léptetése
19     add edi,4
20     sub ecx,1         ; ciklusváltozó léptetése
21     jnz .inner_loop   ; ismétlés, ha van még adat
```

Ami először feltűnhet, az az, hogy a ciklusmag elég rövid: összesen tizenegy gépi utasításból áll és 27 bájtot foglal el a memóriában. C++ nyelven ugyanezt megírva hosszabb gépi kódot kaptunk volna, mert nem tudtuk volna közvetlenül elérni a processzor nyújtotta lehetőségeket. A másik fontos tulajdonsága a kódnak, hogy csak a forrás- és célváltozók eléréséhez használ memóriát, a részeredményeket mind regiszterekben tartja. Ezt x86 processzoron nehéz megvalósítani, mivel csak hét regiszter áll a rendelkezésünkre - a ciklus mind a hetet ki is használja. További megkötés, hogy a MUL utasítás egyik forrása és a célja rögzített (EAX és EDX:EAX), ezért valamelyik operandust mindenképp mozgatnunk kell. Ha a regiszterben lévőt mozgatnánk, az egy órajelig tartana, majd a MUL utasításnak három további órajel kellene a memória olvasására a szorzáson felül. Ezért inkább a memóriában lévőt mozgatjuk, ami három órajelig tart, de cserébe a MUL utasításnak már mindkét operandusa rendelkezésre áll, és összességében egy órajellel hamarabb végzünk, mint az első esetben. Ciklusonként két memóriaolvasás és egy írás történik – egy olvasás az 5. sorban, egy olvasás és egy írás a 13. sorban. (A CISC utasításkészlet sajátossága, hogy egy utasítás egyszerre olvas, számol és ír – ez elősegíti a gépi kód tömörségét, de bonyolítja a dekódolást.) Az architektúra spekulatív végrehajtást alkalmaz, azaz megpróbálja megjósolni a feltételes ugrások irányát, és előbb elkezdi az ugrás utáni kód végrehajtását, mint ahogy a feltétel értéke elérhetővé válik. Ha a jóslás helyes volt, az ugrás költsége minimális, ha viszont helytelen, vissza kell vonni az ugrás utáni uta-

sításokat, ami 10-20 órajelbe is beletelhet. A ciklus végén levő feltételes ugróutasításról azt feltételezi majd a processzor, hogy visszaugrik, így a ciklus utolsó iterációjának végén hibás lesz a jóslás. Hogy ennek hatását minimalizáljuk, érdemes maximalizálni a belső ciklus ciklusszámát azzal, hogy a hosszabb szorzandót dolgozzuk fel a belső ciklusban.

Érdekes módon, a fenti kód teljesítménye egy régebbi 1,7 GHz-es (Willamette magos) Intel Celeron processzoron jócskán elmaradt a várttól, figyelembe véve az órajelek közötti különbséget. Az optimális teljesítményt itt úgy tudtam elérni, hogy a fenti logikát a hagyományos utasítások helyett az SSE2 utasításkészlet tagjaival implementáltam. Az SSE2 az SIMD műveletvégzésre lett kifejlesztve, de speciális esetként lehetséges vele a 64 bit széles regiszterek egy egységként való kezelése. Szorozni ezzel is csak 32 bites egységekben tudunk, de az összeadás és kivonás egy lépésben elvégezhető 64 bitre. Ezen az architektúrán a hagyományos szorzás késleltetése kb. 16 órajel, míg az SSE2 szorzó utasítása csak 6 órajelet vesz igénybe; az SSE2 64 bites összeadása is jóval gyorsabb, mint a két 32 bites összeadás. Ezek miatt még úgy is megéri az új utasításkészlet használata, hogy nem is párhuzamosítunk vele. Az SSE2-re implementált megvalósítás forrása:

```
1  ; az ESI az aktuális forráscímet, az EDI a célcímet tartalmazza
2  ; az MM0 a külsõ ciklus aktuális számjegye, ezzel szorzunk mindent
3  ; az MM2 tartalmazza az elõzõ iterációból maradt carry értéket (32 bit)
4  ; az MM0..MM7 regiszterek 64 bit szélesek
5  .inner_loop:
6      movd mm1,[esi]    ; a forrás számjegy betöltése az MM1-be
7                        ; zéró kiterjesztéssel
8      movd mm3,[edi]    ; a cél számjegy betöltése az MM3-ba
9                        ; zéró kiterjesztéssel
10     pmuludq mm1,mm0   ; az MM0 és MM1 (alsó 32 bitjének) szorzása,
11                       ; az eredmény az MM1-be kerül
12     paddq mm2,mm3     ; a cél számjegy és a szorzat hozzáadása a carry-hez
13     paddq mm2,mm1
14     movd [edi],mm2    ; az eredmény alsó 32 bitjének kiírása
15                       ; a cél számjegybe
16     psrlq mm2,32      ; az eredmény jobbra tolása 32 bittel,
17                       ; hogy csak a carry maradjon benne
18
19     add esi,4         ; forrás- és célcím léptetése
20     add edi,4
21     sub ecx,1         ; ciklusváltozó léptetése
22     jnz .inner_loop   ; ismétlés, ha van még adat
```

A fenti kódot valamivel leegyszerűsíti, hogy a két 64 bites összeadáshoz csak egy-egy utasításra van szükség. Az SSE2 nem teszi lehetővé, hogy közvetlenül a memória tartalmához adjuk hozzá értéket, ezért a cél számjegyet be kell töltenünk regiszterbe. Ez nem okoz problémát, hiszen így is csak négyet használunk ki a rendelkezésre álló nyolc MM regiszterből. Lehetséges lenne egy 128 bites XMM regisztert felhasználva két 32×32 bites szorzást párhuzamosan végrehajtani, de több időt töltenénk el a bemenet megfelelő alakra hozásával és a kimenet visszaalakításával, mint amennyit a párhuzamosított szorzással nyertünk.

A fenti kód durván kétszer olyan gyorsan futott az említett Celeron processzoron, mint a hagyományos utasításokat használó párja; a két modernebb AMD processzoron viszont másfélszer annyi időt vett igénybe. Ezt főképp azzal magyarázhatjuk, hogy a tesztelt AMD architektúrákon ugyanannyi időbe kerül az SSE2-t használó szorzás, mint a hagyományos párja, a szorzáson kívüli utasítások pedig a második megoldásban az időigényesebbek.

5.2.3. Négyzetre emelés

A négyzetre emelés, mint a szorzás speciális esete, természetesen felírható a hagyományos szorzással:

$$X \times X = (x_0 B^0 + x_1 B^1 + \ldots + x_{k-1} B^{k-1})(x_0 B^0 + x_1 B^1 + \ldots + x_{k-1} B^{k-1}) = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} x_i x_j B^{i+j}$$

Ha jobban megnézzük ezt az egyenletet, észrevehetjük, hogy majdnem mindegyik szorzás kétszer szerepel a jobb oldalon: egyszer $x_a x_b$, majd $x_b x_a$ alakban. Ezeket pazarlás lenne kétszer kiszámolni, átírhatjuk hát a szorzást:

$$\sum_{i=1}^{k-1} \sum_{j=1}^{k-1} x_i x_j B^{i+j} = 2 \sum_{i=0}^{k-1} \sum_{j=0}^{i-1} x_i x_j B^{i+j} + \sum_{i=0}^{k-1} x_i^2 B^{2i}$$

Ez alapján a négyzetre emelést három fázisban végezzük el:

1. Kiszámoljuk az „átló alatti” szorzatokat egy módosított szorzóciklussal
2. Az így kapott összeget balra toljuk egy bittel a kettővel való szorzáshoz
3. Kiszámoljuk az „átlót”, azaz a forrásszám számjegyeinek négyzetét, és ezeket egymástól két-két számjegy távolságra hozzáadjuk az eddigi eredményhez

Az egész művelethez $\frac{k(k-1)}{2} + k = \frac{k^2+k}{2}$ db. elemi szorzásra van szükség, ami majdnem feleannyi, mint a hagyományos szorzáshoz szükséges k^2 elemi szorzás. Ez különösen azért fontos, mert a moduláris hatványozáshoz sok négyzetre emelés szükséges, így ezt a műveletet a lehető leggyorsabban kell tudnunk elvégezni.

5.2.4. Karatsuba-féle szorzás

Ez a szakasz kissé rendhagyó lesz – egy olyan algoritmusról szól, amely végül *nem* került bele a programba. Azért említem meg mégis, mert sokáig úgy tűnt, hogy fel lehet majd használni a dekódolás gyorsítására, és csak az implementáció elkészítése és tesztelése után vált egyértelművé, hogy a konkrét architektúrán ehhez a feladathoz nem alkalmas.

A Karatsuba-féle szorzás alapötlete az, hogy két n bites szám szorzását visszavezetjük három $\frac{n}{2}$ bites szorzásra, valamint pár összeadásra és kivonásra[Knuth, 294. o.]. Legyen a két szorzandó X és Y. Mindkettőt kétfelé bontjuk, vagyis $X = x_1 B^{\frac{n}{2}} + x_2$ és $Y = y_1 B^{\frac{n}{2}} + y_2$ alakban írjuk fel. Ekkor a szorzat alakjára a következőt kapjuk:

$$XY = \left(x_1 B^{\frac{n}{2}} + x_2\right)\left(y_1 B^{\frac{n}{2}} + y_2\right) = x_1 y_1 B^n + \left(x_1 y_2 + x_2 y_1\right) B^{\frac{n}{2}} + x_2 y_2$$

Ez önmagában még négy szorzást jelentene, ami nem hozna gyorsulást a hagyományos szorzáshoz képest. Itt lép be az algoritmus alapötlete: az $x_1 y_2 + x_2 y_1$ összeget egyetlen szorzással is ki tudjuk számolni, a következő gondolatmenet szerint:

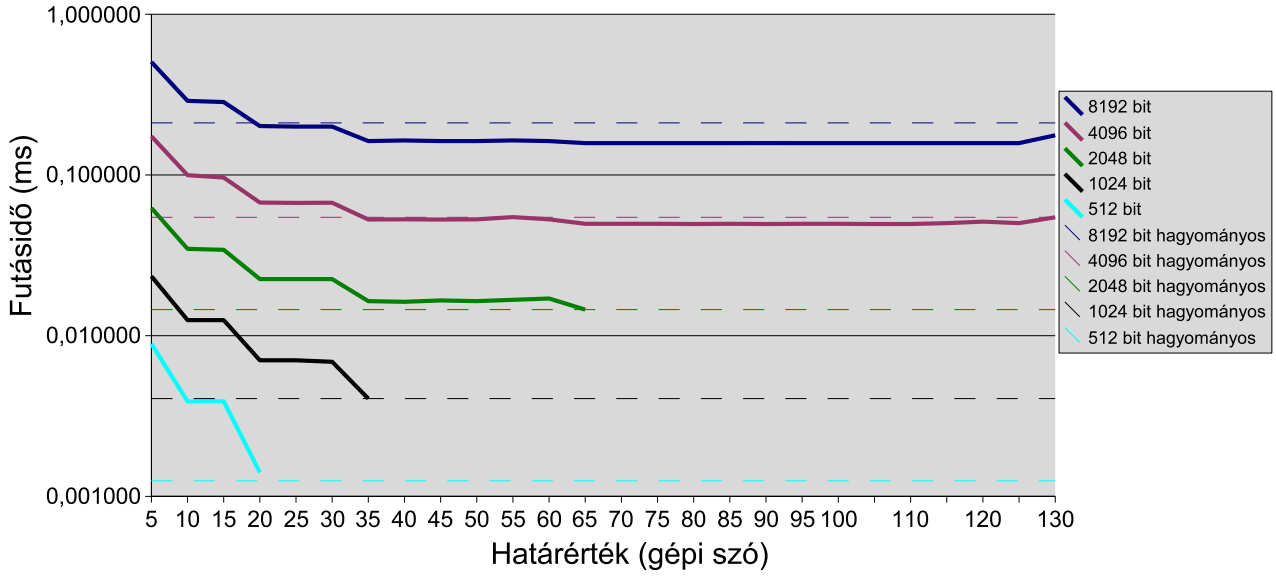
$$\left(x_1 + x_2\right)\left(y_1 + y_2\right) = x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2$$

Az első és az utolsó tagot egyébként is ki kell számítanunk, ezért a középső két tagot megkaphatjuk $\left(x_1 + x_2\right)\left(y_1 + y_2\right) - x_1 x_2 - x_2 y_2$ alakban. A két szorzás és egy összeadás helyett egy szorzást, két összeadást és két kivonást kell végeznünk. Maga a teljes algoritmus a következő lépésekből áll:

1. Számítsuk ki $P = x_1 y_1$ értékét
2. Számítsuk ki $Q = x_2 y_2$ értékét
3. Számítsuk ki $R = \left(x_1 + x_2\right)\left(y_1 + y_2\right) - P - Q$ értékét
4. A végeredmény $XY = PB^n + RB^{\frac{n}{2}} + Q$

A felhasznált három szorzáshoz természetesen használhatjuk rekurzívan magát a Karatsuba-algoritmust. Ha így teszünk, és a bemenetek hosszát a kettő hatványának vesszük, a szorzás futásidejére a következő rekurzív összefüggést kapjuk: $T(n) = 3T(\frac{n}{2}) + 5a + c$ (feltéve, hogy az összeadás és kivonás számjegyenként a időegységbe telik, és c az algoritmus konstans időigénye). A mester tétel alapján ekkor a futásidő $\mathcal{O}(n^{\log_2 3})$. $\log_2 3 \approx 1,5850$, tehát az algoritmus aszimptotikusan hatékonyabb a hagyományos szorzásnál, amely $\mathcal{O}(n^2)$ időbonyolultságú.

Ez persze nem jelenti azt, hogy az algoritmus a gyakorlatban is mindig hatékonyabb lenne a hagyományos szorzásnál. A bemenetek felbontása, az összeadások és a rekurzív



1. ábra. A hagyományos és a Karatsuba algoritmus futásidejének összehasonlítása

hívás költsége miatt egy bizonyos méret alatt gyorsabbá válik az egyszerűbb algoritmus használata. Épp ezért a Karatsuba algoritmus implementációjába mindig beépítenek egy határérték hosszt, ami alatt visszavált a hagyományos módszerre. Ennek a határértéknek a konkrét értéke függ az architektúrától (leginkább az összeadás és a szorzás időigényének arányától), valamint a hagyományos és a Karatsuba szorzás implementációjának minőségétől. Az 1. ábra az általam implementált Karatsuba-szorzás és hagyományos szorzás sebességét mutatja a Karatsuba algoritmus különböző határérték-választása mellett. (A méréseket egy AMD Sempron 2800+ egymagos processzoron hajtottam végre, 32 bites módban. Az időtengely logaritmikus.)

Az ábrából egyből látszik, hogy a Karatsuba algoritmusnak van létjogosultsága, de csak 4096 bites, vagy annál hosszabb operandusokkal, és csak 30-35 gépi szó körüli határértékkel. Az RSA jelenleg használt legnagyobb kulcshossza 4096 bit, amit dekódoláskor két 2048 bites maradékként kezelünk, tehát soha nem fogunk olyan nagy számokat kezelni, ahol a Karatsuba-szorzás gyorsíthatna a jelenlegi implementáción. 64 bites módban nincs értelme futtatni a teszteket, mivel ott ugyanaz a bithossz fele akkora szószámot tesz ki, tehát még kevesebb gyorsulást tudnánk elérni az algoritmussal.

5.2.5. Egészosztás hosszú osztóval

A hosszú számok osztása, hasonlóan az írásbeli osztáshoz, bonyolult algoritmust használ: minden iterációban megbecsüli az eredmény következő számjegyét az osztandó és az osztó első számjegyei alapján, visszaszoroz, és ha rossznak bizonyult a becslés, korri-

gálja a részeredményeket. A művelet végén megkapjuk a lefelé kerekített hányadost és a maradékot. Ehhez négyzetes mennyiségű szorzást és lineáris mennyiségű osztást kell végeznünk. A becslési algoritmus matematikai háttere bonyolult, és igazából nem is vág a témánkba, mivel ilyen osztásokat csak algoritmusok előkészítéséhez kell használnunk, a teljesítményük nem lényeges. Az algoritmus részletes leírása és elemzése megtalálható a hivatkozott irodalomban [Knuth, 272. o.].

5.2.6. Egészosztás rövid osztóval

A prímgenerálás második szűrőjében használjuk ezt a speciális osztást, amikor 65536 alatti prímszámokkal végzünk próbaosztást. Az algoritmus annyi elemi osztást végez, amennyi számjegyből áll az osztandó, de csak akkor használható, ha az osztó egyetlen számjegyből áll.

Legyen az osztandó $X = x_0 + x_1 B + \ldots + x_k B^k$, az osztó pedig $0 < y < B$. Az algoritmust azzal kezdjük, hogy osztjuk x_k-t y-nal: a hányados legyen h, a maradék pedig m. (Az x86-os osztás mindkét értéket visszaadja, ezért nem csökkenti a teljesítményt, ha mindkettőt felhasználjuk.) Ez az $X/(yB^k)$ elvi osztásnak felel meg: ennek a hányadosa szintén h, a maradéka pedig $M = x_0 + x_1 B + \ldots + x_{k-1} B^{k-1} + mB^k$. A h érték lesz a hányados legfelső számjegye. Az M legfelső két számjegyét újból oszthatjuk y-nal. ($m < y$, ezért az $(mB + x_{k-1})/y$ osztás garantáltan nem fog túlcsordulni.) Ez adni fog egy új számjegyet a hányadosból, és egy új maradékot, amivel folytathatjuk az eljárást. Ezt egészen a legalsó helyi értékig folytatjuk, amikor a maradék $x_0 + mB$ értéket osztjuk y-nal. Ennek az utolsó osztásnak a hányadosa lesz a hosszú hányados legalsó számjegye, a maradék pedig az osztás végső maradéka. Figyeljük meg, hogy a közbülső M maradékértékeket nem kell tárolni – elegendő csak feljegyezni az előző osztás maradékát, és „hozzáolvasni" a soron következő számjegyet az osztandóból. (Az x86-os architektúra ezt még meg is könnyíti azzal, hogy a maradékot abba a regiszterbe számolja ki, amely az osztandó felső helyi értékű 32 bitjét tárolta, így a maradékot még csak át sem kell helyezni másik regiszterbe.) Ha csak a maradék érdekel minket, a hányados számjegyeit nem is kell tárolni.

A konkrét megvalósításban még alkalmaztam egy optimalizációt, kihasználva, hogy mindig ugyanazokkal a számokkal végezzük az osztást. A vizsgálandó számot nem az egyes prímekkel, hanem azok szorzataival osztottam, a lehető legtöbb prímet összeszorozva úgy, hogy a szorzat még elférjen a gépi szóban. Miután megkaptam ennek az osztásnak a maradékát, ezt az egy gépi szó hosszú maradékot osztottam el az egyes prímekkel. Ezzel elértem, hogy egy hosszú osztás ideje alatt több osztóval is tesztelni tudjak. Például az első hét, ötnél nagyobb prímszám szorzata még elfér 32 biten, és nyilván ezen prímekkel lehet „megbuktatni" a legtöbb számot; a számok legtöbbje tehát már az első osztási menet után

kiszűrhető. Ez azért fontos, mert az x86-os architektúra osztása nagyon lassú művelet, akár tizenháromszor olyan lassú, mint a szorzás.

5.2.7. Maradékszámítás fix, hosszú osztóval (Barrett-redukció)

Mint láttuk, a moduláris hatványozáshoz minden lépés után szükséges egy maradékszámítás. Erre tudunk a hagyományos hosszú osztási algoritmusnál hatékonyabb algoritmust adni, ha kihasználjuk, hogy az osztó mindig ugyanaz, ezért megéri egy időigényesebb előszámítási fázist beiktatni, ha az csökkenti a tényleges művelet időigényét. A Barrett-redukció egy ilyen algoritmus, amely azt használja ki, hogy az osztás megfeleltethető a reciprokkal való szorzásnak[Handbook, 603. o.]. Ez ebben a formában persze csak akkor igaz, ha a reciprokot teljes pontosságban tároljuk, ami az esetünkben nem lenne kifizetődő, arról nem is beszélve, hogy a végtelen szakaszos bináris törtek kezelése megnehezítené a feladatunkat. Az x szám reciprokjának közelítéséhez az $\tilde{r} = \left\lfloor \frac{B^{2n}}{x} \right\rfloor$ egészet használjuk, ahol n olyan, hogy $B^{n-1} \leq x < B^n$. Bizonyítható, hogy ebben az esetben minden $a < x^2$ számra igaz, hogy a $\tilde{q} = \left\lfloor \frac{a\tilde{r}}{B^{2n}} \right\rfloor$ közelítés legfeljebb eggyel kisebb, mint a valódi hányados, $q = \left\lfloor \frac{a}{x} \right\rfloor$. A B hatványaival való (lefelé kerekített) osztás egyszerűen az alsó számjegyek „levágását” jelenti, így az osztást ki tudtuk küszöbölni, viszont cserébe egy $n \times 2n$-es szorzást kell végeznünk, amihez $2n^2$ db. elemi szorzás szükséges. Ezután még vissza is kellene szoroznunk, hiszen a maradékra vagyunk kíváncsiak, nem a hányadosra. Látható, hogy ebben a formában még lassabb lenne a művelet, mint a hagyományos hosszú osztás.

Első optimalizációként azt vehetjük észre, hogy a $\tilde{q}$ kiszámításakor az $a\tilde{r}$ szorzat alsó $2n$ számjegyét egyből eldobjuk. Jó lenne olyan szorzási módszert találni, ami ki sem számítja ezeket az alsó számjegyeket. Az átvitel miatt nem tudunk olyan algoritmust készíteni, amely a felső számjegyeket pontosan meg tudná adni az alsók kiszámítása nélkül, hiszen akár a legkisebb helyi értékű szorzat is okozhat olyan átvitelt, ami a legfelső helyi értékű számjegyet megváltoztatja. Persze az is látható, hogy az ilyen átvitelek csak nagyon ritkán következnek be. Tudunk olyan közelítést adni a felső számjegyekre, amely az esetek túlnyomó részében pontos lesz, és pontatlan esetben sem lesz egynél nagyobb a hiba. Legyen $X = x_0B^k + x_1B^{k-1} + \ldots + x_k$ és $B = y_0B^l + y_1B^{l-1} + \ldots + y_l$! (A számjegyeket csökkenő helyi érték szerint számoztuk, mert így kényelmesebb felírni a becslést.) Ekkor $M = X \times Y$ szorzat felső p számjegyét a következőképpen becsülhetjük:

$$\tilde{M} = \sum_{i+j \leq p} x_i y_j B^{(k-i)+(l-j)} = M - \sum_{i+j > p} x_i y_j B^{(k-i)+(l-j)}$$

Látható, hogy a legfelső helyi érték $k+l+1$ (az x_0y_0 felső számjegye), az utolsó kiszámított helyi érték a $k+l-p$, vagyis kettővel több számjegyet számítunk ki, mint amennyi pontos számjegy kell. A második alakból kiindulva becsülhetjük a hibát. Kezdjük a legkisebb

hibataggal. Bár pontosan egy ilyen tag lesz, az általánosság kedvéért vegyük úgy, hogy a lehető legtöbb, $t = \min(k+1, l+1)$ db. ilyen tag van. (Legfeljebb annyiféle szorzat létezhet, ahány értéket a kisebbik indextartomány felvehet. Azért kell egyet hozzáadni, hogy a nullás indexet is beleszámoljuk.) Egy elemi szorzat értéke legfeljebb $(B-1)^2$ lehet, ezért a legkisebb helyi értéken levő tagok összege legfeljebb $t(B-1)^2$ lehet. Az eggyel nagyobb helyi értéken hasonló gondolatmenet szerint $t(B-1)^2B$ lehet az összeg. Tegyük most fel, hogy $t+1 \leq B$! Ha ez igaz, akkor $t(B-1)^2 \leq B(B-1)^2$, így az alsó két helyi értéken a tagok összege legfeljebb $(t+1)(B-1)^2B$. A harmadik legkisebb helyi értéken $t(B-1)^2B^2$ a tagok összege; $(t+1)(B-1)^2B \leq (B-1)^2B^2$, ezért az alsó három helyi értéket összesen $(t+1)(B-1)^2B^2$ alakban becsülhetjük. Innen már látszik, hogy ezt a becslést tetszőlegesen kiterjeszthetjük a felsőbb helyi értékekre is, és az összes hibatag összegére a $(t+1)(B-1)^2B^{k+l-p-1}$ felső becslést adhatjuk. A becslést rontsuk még kicsit $(t+1)B^{k+l-p+1}$ alakra; így már láthatóvá válik, hogy a hiba legrosszabb esetben a kívánt pontosság után egy helyi értékkel jelentkezik. Itt két eset lehetséges: ha a $k+l-p+1$ helyi értéken levő számjegy kisebb, mint $B-t-2$, akkor a legnagyobb hiba esetén sem lehetséges túlcsordulás az „értékes" számjegyekbe, és a végeredményünk pontos. Ha viszont a számjegy nagyobb vagy egyenlő a fenti értékkel, a végeredményünkről csak annyit állíthatunk, hogy legfeljebb eggyel kisebb, mint a valós érték (amennyiben lett volna túlcsordulás). Ebben az esetben a pontos végeredményhez növelni kell a szorzás precizitását, legrosszabb esetben teljesen végig kell számolni az összes számjegyet.

Látjuk, hogy a hiba valószínűsége legfeljebb $\frac{\min(k,l)+1}{B}$; 32 bites esetben $B > 4 \times 10^9$, vagyis még ezer számjegy hosszú operandusok esetén is egy a millióhoz nagyságrendű. A $t+1 < B$ korlátozás nem jelent problémát esetünkben; túllépéséhez mindkét operandusnak legalább négymilliárd számjegyből kellene állnia, ezek pedig már be sem férnének az elvileg lehetséges 4 GB memóriába. 64 bites esetben a hiba valószínűsége még elenyészőbb, és a hosszkorlát még kevésbé okoz problémát. Az $\tilde{M}$ kiszámításához legfeljebb $\frac{(p+1)(p+2)}{2}$ db. elemi szorzásra van szükség, ami durván feleannyi, mint egy hagyományos $p \times p$ bites szorzás időigénye.

Elő tudjuk tehát állítani a $\tilde{q}$ becslését, $\hat{q}$-ot, legfeljebb 1 hibával, és ehhez kb. $\frac{n^2}{2}$ szorzást használtunk fel. Mivel a $\tilde{q}$ maga is a q hányados becslése volt 1 hibával, igaz lesz, hogy $q-2 \leq \hat{q} \leq q$. Mi persze nem a hányadosra, hanem a maradékra vagyunk kíváncsiak. Megtehetnénk, hogy visszaszorzunk $\hat{q}$ értékével, majd kivonással kapjuk az m maradékot (illetve $m \leq \tilde{m} \leq 2x+m$ értéket, amiből legfeljebb két kivonással kapható m), ez viszont megint csak túl sok időbe telne, és még mindig alulmaradnánk a hagyományos osztással szemben. Szerencsére ezt elkerülhetjük; ha $B > 3$, a visszaszorzás eredményének elegendő csak az alsó $n+1$ számjegyét ismerni, mert ennek ismeretében is meghatározható a maradék. Először lássuk be, hogy a visszaszorzás utáni különbség, $\tilde{m}$, elfér $n+1$

számjegyen:

$$\tilde{m} = a - \hat{q}x = qx + m - \hat{q}x = (q - \hat{q})x + m \leq 2x + m \leq 2B^n + B^n < B^{n+1}$$

Ha a különbség $n+1$ számjegy hosszú, akkor két eset lehetséges:

1. Nem volt átvitel az $n+2$ helyi értékre, a felső számjegyek mind egyeztek. Ugyanezt az eredményt fogjuk kapni, ha csak az alsó $n+1$ számjegyen végezzük a kivonást.

2. Volt átvitel az $n+2$ helyi értékre, a felső számjegyek alkotta szám eggyel nagyobb a kisebbítendőben, mint a kivonandóban. Ha az alsó $n+1$ számjegyen végezzük a kivonást, az átvitel miatt $a - \hat{q}x - B^{n+1}$ értékét fogjuk kapni.

A két eset között a végeredmény előjele alapján egyértelműen tudunk dönteni, és a második esetben az eredményhez hozzá tudjuk adni B^{n+1} értékét a korrigáláshoz. Ezután igaz lesz, hogy $m \leq \tilde{m} \leq 2x + m$, vagyis $\tilde{m}$-ból addig kell kivonni x-et, amíg az eredmény x-nél kisebb nem lesz.

Már csak az maradt hátra, hogy hatékonyan ki tudjuk számítani $\hat{q}x$ alsó $n+1$ számjegyét. Az elv hasonló, mint a fent leírt részleges szorzásban, de itt alulról számoljuk a helyi értékeket, és nincs szükség extra számjegyek kiszámítására, mivel itt nincs lentről jövő átvitel. A legfelső helyi érték felé való átvitelt egyszerűen eldobjuk. Az elemi szorzások száma $\frac{p(p+1)}{2}$, megint csak durván a fele a hagyományos szorzás igényének.

Összefoglalva az algoritmust:

1. (előszámítás) Számítsuk ki $\tilde{r} = \frac{B^{2n}}{x}$ értékét

2. Számítsuk ki $\hat{q}$-ot, $a\tilde{r}$ felső n számjegyét

3. Számítsuk ki a'-t, $\hat{q}x$ alsó $n+1$ számjegyét

4. Számítsuk ki $\tilde{m} = (a \bmod B^{n+1}) - a'$ értékét. Ha negatív lett, adjunk hozzá B^{n+1}-t

5. Amíg $\tilde{m} > x$, vonjunk ki belőle x-et

6. $\tilde{m} = m$, készen vagyunk

Az algoritmushoz két „fél” szorzást, legfeljebb egy összeadást és legfeljebb három kivonást használtunk fel. Bár a két „fél” szorzás egy kicsit több időbe kerül, mint egy teljes szorzás, az elemi osztások elkerülése miatt mégis hatékonyabb az algoritmus, mint a hagyományos hosszú osztás.

5.2.8. Montgomery-redukció

A moduláris hatványozás szorzó és négyzetre emelő lépései után mindig maradékot számoltunk, hogy elkerüljük a részeredmények hosszának exponenciális növekedését. Ha meggondoljuk, ehhez nem *feltétlenül* csak a maradékszámítás alkalmazható – ha lenne olyan algoritmusunk, ami megtartja az m-mel vett maradékra vonatkozó információt, és képes a bemenete hosszát m hosszához közel tartani, az is megfelelne a célunkhoz. A Montgomery-redukció egy ilyen művelet, amely a Barrett-redukciónál kevesebb elemi szorzást végez, így hatékonyabb hatványozást tesz lehetővé. Ez az algoritmus is egy rögzített m osztót feltételez, valamint felhasznál egy rögzített $R > m$ segédértéket is, amely m-hez képest relatív prím kell legyen. A bemenetként kapott $0 \leq a < mR$ számból képes kiszámítani az $aR^{-1} \bmod m$ értékét, feltéve, hogy hatékonyan tud osztani R-rel[Handbook, 600. o.]. A hatékony osztás megoldható, ha R a B valamely hatványa; kézenfekvő választás az $R = B^n$, ahol $B^{n-1} \leq m < B^n$. A relatív prím feltétel ekkor úgy módosul, hogy m és B relatív prímek kell legyenek; bináris számítógép esetén ez azt jelenti, hogy m csak páratlan lehet. Szerencsére az RSA algoritmus mindig páratlan osztókat használ, így ez nem okoz problémát.

A moduláris hatványozás algoritmusa annyiban módosul, hogy $x \bmod m$ alakú részeredmények helyett $xR \bmod m$ alakúakat használunk. A szorzás vagy négyzetre emelés után maradékszámítás helyett Montgomery-redukciót alkalmazunk: $aR \bmod m$ és $bR \bmod m$ szorzatának Montgomery-redukciója $(abR^2)R^{-1} \bmod m = abR \bmod m$, vagyis a végeredmény alakja megfelel a kiinduló értékek alakjának.

A Montgomery-redukció három lépésre osztható:

1. (előszámítás) $m' = -m^{-1} \bmod R$
2. $k = am' \bmod R$
3. $r = (a + km)\,/R$

Először lássuk be, hogy a harmadik lépésben az osztás nem veszít biteket: az R-rel vett maradék: $a+km \equiv a+am'm \equiv a-am^{-1}m \equiv 0 \pmod R$. Másfelől látszik, hogy a-hoz az osztás előtt m többszörösét adtuk, ami nem változtathatja meg az m-mel vett maradékot, így igaz lesz, hogy $r \equiv aR^{-1} \pmod m$. Még azt kell bizonyítanunk, hogy a méret tényleg csökken: tudjuk, hogy $k < R$ és $a < mR$, így $a + km < mR + Rm = 2Rm$, vagyis az R-rel való osztás után legfeljebb $2m$ lehet az eredmény. Ha r-et m-nél nagyobbnak kapjuk, ki kell vonnunk belőle m-et, hogy elkerüljük a felesleges hossznövekedést. Az egész művelethez (az előszámítást leszámítva) két hosszú szorzásra volt szükségünk: a k alsó számjegyeinek kiszámításához egy részleges szorzásra (kb. $\frac{n^2}{2}$ elemi szorzással) és a km értékéhez egy hagyományos szorzásra (kb. n^2 elemi szorzással). (Az R-rel való osztás

egyszerűen az alsó számjegyek eldobását jelenti.) Ez összesen $\frac{3}{2}n^2$ elemi szorzás, ezzel még a Barrett-redukció alatt maradnánk.

A teljesítmény javításához további „trükköt" kell bevetnünk: ki kell használnunk, hogy a hosszú szorzás kisebb elemi lépésekből áll, így k direkt kiszámítása nélkül is meg tudjuk kapni $a + km$ értékét, a következő módon:

1. (előszámítás) $m' = -m^{-1} \bmod B$. (Figyeljük meg, hogy itt már csak B-vel kell venni a maradékot, nem R-rel!)

2. $r = a$. (r számjegyeit alulról felfelé haladva jelöljük $r_0, r_1 \ldots r_{2n-1}$ alakban. Ebben a lépésben még $r_n = r_{n+1} = \ldots = r_{2n-1} = 0$)

3. Ciklus $i = 0$-tól $n - 1$-ig:

   (a) $u_i = r_i m' \bmod B$. (A maradékképzés gyakorlatilag a felső számjegy eldobását jelenti.)

   (b) $r = r + u_i m B^i$

4. $r = r/R$

5. Ha $r > m$, $r = r - m$

A harmadik lépés mindig m többszörösét adja r-hez, ezért az m-mel vett maradék nem változik a negyedik lépésig. A 3a lépés $u_i \equiv r_i m' \equiv -r_i m^{-1} \pmod B$ értéket számolja ki. A 3b lépés az r_i értékéhez $u_i m \equiv -r_i m^{-1} m \equiv -r_i \pmod B$ értéket adja, vagyis a végeredmény nulla lesz. Mivel ezt n-szer ismételjük, a harmadik lépés után az alsó n helyi értéken végig nullák lesznek, ami azt jelenti, hogy r osztható $B^n = R$ értékével. A harmadik lépés összesen $\sum_{i=0}^{n-1} u_i m B^i$ értéket ad az r-hez; ezt felülről becsülhetjük $(B-1)m \sum_{i=0}^{n-1} B^i = (B-1)m\frac{B^n-1}{B-1} < mB^n = mR$ alakban. Tudjuk továbbá, hogy $a < mR$, így a negyedik lépés előtt $r < 2mR$. Látjuk tehát, hogy bár alig hasonlít az algoritmus az előzőhöz, mégis pontosan ugyanazt a végeredményt szolgáltatja. Az elemi szorzások mind a harmadik lépésben történnek, egy iterációban $n+1$ darab; mivel a törzs n-szer fut, összesen $n(n+1)$ elemi szorzásunk lesz. Sikerült tehát olyan redukciós műveletet kapnunk, ami alig lassabb egy hagyományos $n \times n$-es szorzástól, és biztosan gyorsabb a Barrett-redukciótól. A 3b lépés gyakorlatilag ugyanazt végzi, mint a hosszú szorzás belső ciklusa, csak más paraméterekkel, ezért nagyon könnyű a hosszú szorzás assembly kódját adaptálni a Montgomery-redukcióhoz. Ráadásul nincs szükség az előszámítási lépésben hosszúszámos lnko műveletre – elég a hagyományos euklideszi algoritmust futtatni az m legalsó számjegyére.

Meg kell jegyeznünk azonban, hogy a Montgomery-redukció előtt mindig megfelelően elő kell készíteni az operandusokat (szorozni R-rel modulo m), majd a redukció végeztével vissza kell őket alakítanunk (szorozni R^{-1}-nel modulo m). Mindkettő elvégezhető Montgomery-redukcióval: az előkészítéshez szoroznunk kell R^2 mod m értékével, majd Montgomery-redukciót végezni; a visszaalakításhoz elegendő egy Montgomery-redukció. Az első kb. $2n^2$, a második kb. n^2 idejű művelet, és ezekkel együtt már jóval több időt használunk fel, mint Barrett-redukció esetében. Ez azt jelenti, hogy egy-két maradékképzés esetén nem érdemes Montgomery-redukciót használni; csak akkor lesz hatékony, ha sok redukciót tudunk végezni, ezzel amortizálva az előkészítés és a visszaalakítás idejét. A moduláris hatványozásnál ez nem jelent problémát, hiszen a legjobb esetben is annyi redukciót kell végeznünk, ahány bites a kitevő, és e mellett tényleg eltörpül az előkészítés és a visszaalakítás ideje.

Létezik egy Montgomery-szorzás nevű algoritmus is, amely egy menetben ötvözi a hagyományos szorzás és a Montgomery-redukció lépéseit[Handbook, 602. o.]. Ez $2n(n+1)$ darab elemi szorzást igényel, pontosan n-nel többet, mint a külön végzett hagyományos szorzás és Montgomery-redukció. Ezt csak akkor lenne érdemes használni, ha a függvényhívásnak nagy költsége lenne, vagy korlátozva lenne a gépi kód mérete. Esetünkben egyik sem áll fenn, ezért a program nem alkalmazza ezt az algoritmust.

5.3. Az algoritmusok teljes időigénye

Most, hogy láttuk, milyen algoritmusok dolgoznak a magas szintű algoritmusok „mögött", tudjuk a legfontosabb magas szintű algoritmusok futásidejét becsülni az általuk elvégzett elemi szorzások számával.

A moduláris hatványozásnál nézzük a „t-szeres" változatot. (A „csúszó ablak" változat futásidejének kiszámítása jóval bonyolultabb lenne, ezért inkább fogadjuk el a „t-szeres" változat futásidejét felső becslésként.) Az 5.1.1 szakaszban már beláttuk, hogy l darab négyzetre emelésre és $\frac{l}{t}$ darab szorzásra van szükségünk, ahol l a kitevő bitjeinek száma, t pedig szabadon választható, az „ablak" bitjeinek száma. Mind a szorzások, mind a négyzetre emelések után szükségünk van egy Montgomery-redukcióra. Az 5.2.2 szakasz alapján a szorzás futásideje n^2, az 5.2.3 szakasz alapján a négyzetre emelés ideje $\frac{n(n+1)}{2}$, és az 5.2.8 szakasz szerint a Montgomery-redukció $n(n+1)$ elemi szorzást igényel, ahol n a tényezők számjegyeinek száma. (Az egyszerűség kedvéért az összes operandust az alappal egyenlő hosszúnak vesszük, bár lehetnek rövidebbek is.) A teljes futásidő így $\frac{3}{2}l(n^2+n)+\frac{l}{t}(2n^2+n)$, vagyis a kitevő hosszának lineáris függvénye, az osztó hosszának pedig négyzetes függvénye. Ha a 3.1 pontban leírt első dekódolási algoritmus alapján két moduláris hatványozást végzünk egy helyett, mind az osztó, mind a kitevő hossza kb. a felére csökken a második algoritmushoz képest, mivel p és q egy nagyságrendben vannak,

valamint $d_1 < p$ és $d_2 < q$. A kitevő hosszcsökkenése kétszeres gyorsulást idéz elő, az osztóé durván négyszereset, de két hatványozást kell végeznünk, ezért az összes gyorsulás négyszeres. Ha van lehetőségünk a két hatványozást két processzormagon párhuzamosan végezni, akkor el tudjuk érni a nyolcszoros gyorsulást is. Láthatjuk tehát, hogy tényleg igaz volt a 3.1 szakasz azon állítása, hogy az első algoritmus hatékonyabb a másodiknál.

A prímgenerálás első szűrője (kerékmódszer) csak összeadásokat használ, így időigénye elhanyagolható a többi fázishoz képest. A második szűrő (próbaosztás) sebességéről már volt szó az 5.2.6 szakaszban: egy próbaosztás a tesztelendő szám hosszával egyenlő számú elemi osztást igényel. A számok túlnyomó többsége az első próbaosztás után megbukik, viszont a prímeken összesen 3256 tesztet fogunk elvégezni 32 bites esetben, vagy 1577-et 64 bites esetben. A Miller-Rabin teszt a probabilisztikus jelleg miatt szintén változó időbe telhet. Egy adott a számmal a tesztelés ideje gyakorlatilag egyezik az $a^{p-1} \bmod p$ moduláris hatványozás idejével, bár a hatványozás utolsó néhány négyzetre emelését a moduláris hatványozó függvényen kívül végezzük. A prímként elfogadott számoknak 30 ilyen teszten kell átmenniük. A kulcshoz szükséges két prím generálása párhuzamosítható több processzormag esetén, de a moduláris hatványozással ellentétben itt nem garantálhatjuk, hogy a két számítás durván azonos időt vesz igénybe; csak annyit mondhatunk, hogy párhuzamosított esetben a kulcsgenerálás addig tart, mint a hosszabb ideig tartó prím generálása.

6. Gépközeli optimalizáció

Az optimalizáció előtt mindig nagyon fontos a program futásának elemzése profiler program segítségével, valamint a „forró pontok” azonosítása. Ezek olyan kódsorok, amelyek végrehajtása a futásidő jelentős részét teszi ki. Azonosításuk azért fontos, hogy tudjuk, a program melyik részére kell összpontosítanunk: egy „forró pont” 10%-os gyorsítása sokkal jobban megéri, mint egy ritkán futó rész tízszeres gyorsítása.

Az AMD CodeAnalyst profiler program adatai szerint ha előre generált kulccsal indulva, csak titkosítást és visszafejtést végzünk (2048 biten), az idő 55,3%-ában Montgomery-redukciót végez a program, 28,6%-ot tesz ki a négyzetre emeléssel töltött idő, és 8,8%-ot a hagyományos szorzás ideje. Összesen a futásidő 92.9%-át töltjük ebben a három kis méretű rutinban. Ha csak kulcsot generálunk (szintén 2048 biten), akkor az arányok a következőképpen alakulnak: 43,8% Montgomery-redukció, 23,9% négyzetre emelés, 18,7% próbaosztás és 7% hagyományos szorzás (összesen 94% ebben a négy rutinban). Láthatjuk, hogy ez a négy művelet alkotja a program „forró pontjait”; a Montgomery-redukció kitüntetett figyelmet érdemel, mivel az idő több, mint felében fut. Ezeket mindenképpen megéri assembly nyelven implementálni, hiszen majdnem minden egyes megtakarított óra-

jel kimutatható teljesítménynövekedést jelent. A programomban ezen a négy műveleten felül még kettő szerepel assembly nyelven: a Barrett-redukcióhoz használt „alsó” és „felső” részleges szorzás. Ezek a hagyományos szorzás kódjából kis változtatással megkaphatók, így elkészítésükhöz nem volt szükség nagy erőfeszítésre.

Ahhoz, hogy az assembly nyelvű implementáció tényleg gyorsabb legyen a C++ fordító által generáltnál, elengedhetetlen az utasításkészlet részletes ismerete (az egyes utasítások időigényét is beleértve), valamint a processzorgyártó által rendelkezésre bocsátott optimalizációs irányelvek megértése. A mai processzorok a visszafelé kompatibilitás érdekében megtartottak minden régi utasítást, de ez nem jelenti azt, hogy mindegyiket a lehető legjobb teljesítménnyel tudják végrehajtani. Előfordul, hogy két egyszerű utasítás gyorsabban lefut, mint ha az ugyanazt eredményező komplex utasítást használnánk. A szuperskalár jelleg miatt arra is figyelnünk kell, hogy milyen függőségek állnak fenn az utasítások között – ha egy utasításnak meg kell várnia az előző eredményét, akkor nem lehetséges a párhuzamosítás, és nem érjük el a potenciálisan lehetséges teljesítményt. Érdemes lehet két vagy több művelet elemi lépéseit felváltva végezni, hogy a processzor képes legyen azt „a színfalak mögött” egyszerre végrehajtani. A feltételes ugrások teljesítménycsökkenéssel járhatnak, mert a processzor spekulatívan kezdi el valamelyik ágat végrehajtani, és hibás jóslás esetén ezen műveletek visszavonása időbe telik. Megéri egy bonyolultabb, bitműveleteket használó kóddal kiváltani a feltételes ugrást, ha lehetséges.

Mindezen dolgok miatt egy változtatás hatása sokszor bizonytalan, és ezért elengedhetetlen, hogy minden változtatás után megmérjük a teljesítményt. A mért eredmények gyakran ellentmondanak az intuíciónak, ezért ki kell próbálni az első ránézésre lassabb alternatívákat is. Hasznos ezen kívül minél több processzorral tesztelni – bár az alapelvek azonosak, az implementációs részletek néha nagyon különböznek, és ami az egyik processzoron gyorsít, az a másiknál közömbös lehet, vagy lassíthat. (Az 5.2.2 szakaszban láthattunk erre egy gyakorlati példát.) Nagy, teljesítményigényes projektekben nem ritka, hogy a különböző x86-os processzorcsaládokhoz saját assembly kódot írnak, annak ellenére, hogy azok kölcsönösen kompatibilisek. Erre nekem csak korlátozott lehetőségem volt, mert a fejlesztés nagy részében csak a saját számítógépeimen tudtam tesztelni – így is kétféle assembly nyelvű implementációt kellett írnom, hogy optimális teljesítményt tudjak elérni mindegyik tesztrendszeren.

7. A program teljesítménye

Az alábbi időeredményeket egy AMD Sempron 2800+ egymagos processzoron mértem. Az összehasonlítási alapként az OpenSSL kriptográfiai programcsomag 0.9.8h verzióját használtam [OpenSSL], a forráskódban kissé módosítva, hogy a leggyorsabb implemen-

tációt használja a konstans idejű helyett. Az OpenSSL 64 bites x86-os változata még nem teljesen kiforrott, nem tartalmaz assembly kódú megvalósítást, ezért a teljesítménye elmarad a lehetségestől.

	Saját 32 bit	**OpenSSL 32 bit**	**Saját 64 bit**	**OpenSSL 64 bit**
512 bit kódolás	0,036 ms	0,057 ms	0,016 ms	0,032 ms
512 bit dekódolás	0,496 ms	0,707 ms	0,251 ms	0,478 ms
1024 bit kódolás	0,106 ms	0,149 ms	0,037 ms	0,073 ms
1024 bit dekódolás	2,429 ms	3,077 ms	0,920 ms	1,467 ms
2048 bit kódolás	0,359 ms	0,480 ms	0,120 ms	0,198 ms
2048 bit dekódolás	14,641 ms	16,967 ms	5,088 ms	7,477 ms
4096 bit kódolás	1,302 ms	1,627 ms	0,413 ms	0,623 ms
4096 bit dekódolás	97,057 ms	109,116 ms	32,088 ms	42,717 ms

Az összehasonlítás persze nem teljesen tisztességes, hiszen az OpenSSL többféle titkosítási módot is támogat, valamint több titkosítási „motort” is képes használni, ezek az absztrakciók pedig valamennyi lassulással járnak. Ezen felül az OpenSSL-t többféle architektúrára is le lehet fordítani, míg én csak a rendelkezésemre álló néhány processzorra tudtam optimalizálni. Az azért mindenképp látszik, hogy az én implementációm teljesítménye összemérhető a gyakorlatban használt kriptográfiai megoldások sebességével.

Az időadatokból az is látszik, hogy az RSA még a mostani gépeken sem alkalmas valós idejű titkosításra – még a leggyorsabb, 512 bites dekódolás is csak 129 Kb/s körüli sebességre képes, a legbiztonságosabb 4096 bites dekódolás pedig 5 Kb/s sebességgel hajtható csak végre.

Az adatok jól mutatják még azt is, hogy mennyit számít a gépi szó hossza a hosszúszámaritmetika teljesítményében: az algoritmusok bármilyen változtatása nélkül durván háromszoros teljesítményt voltam képes elérni ugyanazon a processzoron, csupán a gépi szó hosszának megduplázásával. A moduláris hatványozásban a hosszú szorzások és Montgomery-redukciók száma nem függ a gépi szó hosszától, csak a kitevő bitekben mért hosszától; azonban ezen szorzások és redukciók mindegyike negyedannyi elemi szorzással elvégezhető. A 64×64 bites elemi szorzás viszont több időbe kerül, mint a 32×32 bites: az előbbi 5 órajelbe telik, míg az utóbbi csak 3-ba. Ezek alapján azt várhatnánk, hogy a teljesítmény $\frac{5}{3} \times \frac{1}{4} = \frac{5}{12}$-szerese lesz a 32 bitesnek, de úgy látszik, a regiszterek számának duplázása is jelentősen javítja a teljesítményt.

A következő időeredményeket egy AMD Athlon 64 X2 4800+ kétmagos processzoron mértem, 32 bites módban:

	Egy szálon	Két szálon	Futási idők aránya
512 bites dekódolás	0,316 ms	0,247 ms	0,7816
1024 bites dekódolás	1,552 ms	0,928 ms	0,5979
2048 bites dekódolás	9,338 ms	4,859 ms	0,5203
4096 bites dekódolás	61,769 ms	31,468 ms	0,5094

Látszik, hogy bár elméletileg kétszeres gyorsulás érhető el a párhuzamosítással, a gyakorlatban ennél kisebb lesz a teljesítmény a szálak összehangolásának időigénye miatt. Ha maguk a részműveletek elég hosszú ideig tartanak, akkor ez az időigény elhanyagolhatóvá válik, és meg tudjuk közelíteni a kétszeres sebességet.

Az x86-os processzorok közti különbséget jól szemléltetik az alábbi időeredmények, amelyeket az Informatikai Kar két szervergépén mértem, Linux operációs rendszer alatt: Egymagos Intel Xeon 2.4GHz:

	Hagyományos utasításkészlet	SSE2 utasításkészlet	OpenSSL (referencia)
512 bites kódolás	0,0693 ms	0,0387 ms	0,066 ms
512 bites dekódolás	0,9124 ms	0,6494 ms	0,897 ms
1024 bites kódolás	0,2399 ms	0,1168 ms	0,184 ms
1024 bites dekódolás	4,9385 ms	2,7147 ms	4,217 ms
2048 bites kódolás	0,8696 ms	0,3841 ms	0,565 ms
2048 bites dekódolás	33,4351 ms	16,0654 ms	22,67 ms
4096 bites kódolás	3,32668 ms	1,5694 ms	1,790 ms
4096 bites dekódolás	237,826 ms	103, 287 ms	135,676 ms

Kétmagos AMD Opteron 880 (2.4 GHz):

	Hagyományos utasításkészlet	**SSE2 utasításkészlet**	**Hagyományos utasításkészlet (két szálon)**	**OpenSSL (referencia)**
512 bites kódolás	0,0242 ms	0,0333 ms		0,034 ms
512 bites dekódolás	0,3311 ms	0,4008 ms	0,3484 ms	0,471 ms
1024 bites kódolás	0,0714 ms	0,1031 ms		0,093 ms
1024 bites dekódolás	1,6594 ms	2,2141 ms	1,2136 ms	2,069 ms
2048 bites kódolás	0,2391 ms	0,3493 ms		0,308 ms
2048 bites dekódolás	9,8158 ms	13,9682 ms	5,4894 ms	11,375 ms
4096 bites kódolás	0,8679 ms	1,2951 ms		1,055 ms
4096 bites dekódolás	64,7681 ms	95,723 ms	33,2818 ms	73,212 ms

Látható, hogy az SSE2 utasításkészlettel durván kétszeres sebességet lehetett elérni az Intel processzoron, míg ugyanez nagyjából másfélszeres lassuláshoz vezetett az AMD processzorán. Ezen kívül még az tűnhet fel, hogy a linuxos rendszeren nagyobb a szinkronizáció költsége, mint a windowsos tesztrendszeren, ezért kis kulcshossznál kevésbé gyorsít a párhuzamosítás.

8. Összefoglalás

Célom az RSA algoritmus implementálása volt, minél hatékonyabban kihasználva az x86 architektúra nyújtotta lehetőségeket. A fenti sebességadatok fényében azt mondhatom, hogy ez sikerült, igazából felül is múltam a várakozásaimat. A meglevő kódot viszonylag kis erőfeszítéssel fel lehetne készíteni más moduláris hatványozást alkalmazó kriptorendszerek, például az ElGamal rendszer kezelésére is (bár ennek gyors megvalósításához más hatványozási algoritmus az optimális, a szorzó, négyzetre emelő és redukáló algoritmusokat változtatás nélkül felhasználhatjuk). A megvalósítás viszonylag egyszerűen hordozható más architektúrára, bár a jó teljesítmény eléréséhez valószínűleg az új architektúrán

is gépi kódban kellene megvalósítani az alacsony szintű algoritmusok egy részét.

Programom írása során sokat tanultam a programok optimalizációjáról, mind magas, mind alacsony szintű kód esetében. A legfontosabb tanulság az volt, hogy minden változtatás után szükséges a mérés – többször előfordult, hogy egy fontosnak tűnő változás nem hozott gyorsulást, míg egy apró módosítás 4-5%-os teljesítménynövekedést eredményezett. Lényeges annak a megválasztása is, hogy mit implementáljunk magas szintű, és mit alacsony szintű programnyelveken – a túl magas absztrakciós szint lassuláshoz, a túl alacsony nehezen felfedezhető hibákhoz és nehezen karbantartható kódhoz vezet; a kettő között egyensúlyt kell tartani. Az itt szerzett tapasztalatoknak nagy hasznát fogom venni a gyakorlatban előforduló programozási problémák megoldásakor.

Hivatkozások

[Factor-Record] Wikipedia, The Free Encyclopedia: Integer factorization records http://en.wikipedia.org/w/index.php?title=Integer_factorization_records&oldid=197115400

[Rivest-Silverman] Ron Rivest and Robert Silverman: Are 'Strong' Primes Needed for RSA?, Cryptology ePrint Archive: Report 2001/007. http://eprint.iacr.org/2001/007

[Knuth] Donald Knuth: The Art of Computer Programming, 3. kiadás, 2. kötet

[Handbook] Alfred J. Menezes, Paul C. van Oorschot és Scott A. Vanstone: Handbook of Applied Cryptography, 5. kiadás http://www.cacr.math.uwaterloo.ca/hac/

[OpenSSL] Az OpenSSL projekt weboldala http://www.openssl.org/source/

Egyéb felhasznált források

- Intel® 64 and IA-32 Architectures Optimization Reference Manual http://www.intel.com/design/processor/manuals/248966.pdf
- AMD Athlon™ Processor x86 Code Optimization Guide http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf
- Software Optimization Guide for AMD64 Processors http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF
- Agner Fog: Instruction tables - Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel and AMD CPU's http://www.agner.org/optimize/instruction_tables.pdf

Ajánlott irodalom

- Richard E. Crandall. : Projects in Scientific Computation, Springer-Verlag, 1994.
- Dr. Ködmön József, Kriptográfia, Az informatikai biztonság alapjai, ComputerBooks Könyvkiadó, Budapest, 1999.
- Gilles Brassard: Modern Cryptology, Lecture Notes in Computer Science, 325. kötet, Springer-Verlag, 1988.