Univerza v Ljubljani

Fakulteta za elektrotehniko

Matevž Langus

UČINKOVITI ALGORITMI ZA IZVEDBO KRIPTOGRAFSKIH POSTOPKOV

Seminarska naloga pri predmetu
Porazdeljeni informacijski sistemi in celovitost podatkov

Ljubljana, 2006

KAZALO:

1 UVOD 4

2 MATEMATIČNI POSTOPKI V ŠIFRIRNIH ALGORITMIH 6

2.1 Aritmetične funkcije za velika cela števila 6

2.1.1 Seštevanje in odštevanje 7

2.1.2 Množenje 7

2.1.3 Kvadriranje 8

2.1.4 Deljenje 8

2.2 Aritmetika modulov za velika cela števila 9

2.2.1 Modularno seštevanje in odštevanje 9

2.2.2 Klasično modularno množenje 10

2.2.3 Algoritem: Montgomery-jevo množenje 10

2.2.4 Barret-ovo zmanjšanje 11

2.3 Algoritmi za iskanje največjega skupnega deljitelja 11

2.3.1 Binarni gcd algoritem 11

2.3.2 Lehmerjev gcd algoritem 12

2.3.3 Dvojiško razširjeni gcd algoritem 12

2.4 Eksponiranje 13

2.4.1 Univerzalne tehnike eksponiranja 14

2.4.2 Eksponiranje s fiksnim eksponentom 16

2.4.3 Eksponiranje s fiksno osnovo 16

2.4.4 Metoda z oknenjem 17

2.4.5 Evklidova metoda za fiksno osnovo 17

2.4.6 „Comb“ metoda za fiksno osnovo 18

2.5 Prekodiranje eksponenta 18

2.5.1 Zapis z uporabo predznaka 18

2.5.2 Zapis z menjavanjem nizov 19

3 ALTIVEC 21

3.1 Princip delovanja AltiVec 22

3.2 Podatkovni tipi 23

3.3 AltiVec operacije 23

3.4 Pretvorniki tipov 23

3.5 Nastavljanje konstant 24

3.6 Matematične operacije 24

3.7 Logične operacije 25

3.8 Primerjave 25

3.9 Operacije permutacij 26

3.10 Operacije s spominom 26

4 UPORABA ALTIVEC ZA RSA 27

4.1 Ozadje RSA 27
4.2 Uporaba AltiVec za modularno eksponiranje 28
5 ALTIVEC IN KONČNA POLJA **31**
5.1 Seštevanje 31
5.2 Množenje 32
5.3 Zmogljivost 33
6 VIRI **35**

1 UVOD

Današnji in prihodnji varnostni postopki običajno temeljijo na predpostavki, da nek matematični problem, ki je vgrajen kot osnova nekega postopka, predstavlja dovolj velik problem, da ga z obstoječimi in prihodnjimi računskimi pripomočki ni mogoče razrešiti in na ta način razbiti določen varnostni postopek. Za nekatere matematične probleme je dokazano, da ne obstaja enostavne rešitve, za druge pa se trenutno to samo domneva, ker nihče ne more dokazati nasprotno (primer RSA Contest).

Zato na splošno velja, da mora varnostni postopek upoštevati načelo, da naj bo izračunati inverzno funkcijo, ki bi nas pripeljala do šifrirnega ključa, izredno težko, če že ne nemogoče. Pri tem se načrtovalci varnostnih postopkov zanašajo na sledeče matematične probleme:

- problem faktorizacije velikih celih števil,
- problem RSA,
- problem kvadratičnega residuuma,
- problem ostankov kvadratnega korena,
- problem diskretnega logaritma,
- problem Diffie-Hellman,
- problem podvsote celih števil,
- itd.

Ti problemi za trenutno računalniško tehnologijo predstavljajo nepremagljivo oviro, zato lahko varnostne postopke smatramo kot varne. V kolikor pa bi se pojavila drugačna tehnologija reševanja teh problemov, bi vsi varnostni postopki lahko kaj hitro postali ranljivi.

Varnostni postopki morajo kot rečeno zagotavljati, da je inverzna funkcija težko izračunljiva, hkrati morajo uporabljati tako osnovno funkcijo, ki je za trenutno tehnologijo čim lažje izračunljiva, sicer postanejo ti postopki neuporabni. To se lahko zgodi zato, ker za šifriranje sporočila potrebujejo:

- preveč časa,
- preveč virov (pomnilnik, število logičnih vrat, …).

Glede na to, da se danes varnostni postopki uporabljajo v vseh komunikacijskih napravah, od velikih strogo-namenskih šifrirnih sistemov pa vse do enostavnih osebnih prenosnih telefonov, je še kako pomembno, da uporabljeni algoritem ni preveč kompleksen, kljub temu pa zagotavlja zadovoljivo stopnjo varnosti. Konec koncev se taka zahteva pozna lahko tudi na velikosti naprave in dolžini avtonomije delovanja take naprave. Ta problem je opazen pri povsem običajnih vsakodnevnih sistemih, na primer varnem elektronskem bančništvu. Da bi prihranili procesorsko moč, se uporabljata dva tipa šifrirnih postopkov:

- asimetrični za izmenjavo simetričnega ključa,
- simetrični za izmenjavo pravih podatkov.

Znano je, da so simetrični postopki izvedbeno precej manj zahtevni kot asimetrični.

V nadaljevanju si bomo ogledali računske probleme, ki tipično nastopajo pri šifrirnih algoritmih. Glede na to, da so smernice razvoja v svetu take, da se večina postopkov izvede v programski opremi, si bomo na koncu ogledali še možnost uporabe pospeševalnih procesorskih ukazov AltiVec uporabljenih v procesorjih PowerPC, kajti znano je, da običajni procesorji niso ravno optimalni za izvajanje šifrirnih postopkov.

2 MATEMATIČNI POSTOPKI V ŠIFRIRNIH ALGORITMIH

Večina sodobnih šifrirnih postopkov, digitalnih podpisovalnih mehanizmov in zgoščevalnih funkcij uporablja enake osnovne matematične računske operacije. Večinoma se kot algebraično orodje uporabljajo končna polja ostankov pri deljenju celih števil (Z_m, kjer je m veliko celo pozitivno število). Najbolj poznani, kot so RSA, Rabin in ElGamal uporabljajo množenja in eksponiranja v Z_m. Čeprav se večinoma uporablja Z_m, so aktualne tudi ostale strukture: polinomski obroči, splošna končna polja, končne krožne skupine, … Kot primer naj navedem skupino točk, ki ležijo na eliptični krivulji in se nahajajo znotraj končnega polja.

Učinkovitost katerekoli računske strukture zavisi od več dejavnikov: velikost parametrov, procesna moč, potratnost pomnilnika, programske ali strojne optimizacije, tip matematičnega algoritma, … Ogledali si bomo matematične postopke za izvajanje najbolj tipičnih operacij (seštevanje, odštevanje, množenje, deljenje in eksponiranje) za najbolj tipično algebraično strukturo Z_m.

Pri ocenjevanju algoritmov za izvajanje določene operacije je potrebno upoštevati, da za isto operacijo obstaja več algoritmov, ki so bolj ali manj optimalni za dano arhitekturo. Odvisno od tega, kje želimo algoritem implementirati, je lahko bolj pomembna procesna moč in čas izvajanja, spet drugje pa je bolj pomembno, da algoritem zaseda čim manj pomnilnika tako za kodo, kot za podatke, ki jih uporablja za delo.

Ogledali si bomo naslednje operacije:

- aritmetične funkcije za velika cela števila
- funkcije ostanka za velika cela števila
- algoritmi za iskanje največjega skupnega deljitelja
- eksponiranje
- prekodiranje eksponenta

2.1 Aritmetične funkcije za velika cela števila

V šifrirnih postopkih se pogosto uporabljajo velika cela števila, kar se nam zdi povsem normalno, žal pa računalnikom to predstavlja problem. Predvsem pri manjših procesorjih je ta problem dokaj hitro opazen. Na primer pri 16-bitnem procesorju so vse aritmetične operacije, ki jih procesor zmore izvajati 16 bitne. Z drugimi besedami, največje vrednosti, ki jih procesor še zmore obdelovati so do 65536. 32-bitni procesorji, ki so v računalnikih najbolj razširjeni, lahko računajo s števili do $4*10^9$. Tudi te vrednosti so premajhne za uporabo v šifrirnih algoritmih, zato je potrebno razviti posebne algoritme za popolnoma osnovne funkcije kot so seštevanje, množenje, …

2.1.1 Seštevanje in odštevanje

Seštevanje in odštevanje se lahko izvaja med dvema številoma enake dolžine. Če je eno število krajše, se mu spredaj doda še toliko ničel, da postane enake dolžine.

Algoritem seštevanja:

INPUT: positive integers x and y, each having $n + 1$ base b digits.
OUTPUT: the sum $x + y = (w_{n+1} w_n \cdots w_1 w_0)_b$ in radix b representation.

1. $c \leftarrow 0$ (c is the *carry* digit).
2. For i from 0 to n do the following:
   2.1 $w_i \leftarrow (x_i + y_i + c) \bmod b$.
   2.2 If $(x_i + y_i + c) < b$ then $c \leftarrow 0$; otherwise $c \leftarrow 1$.
3. $w_{n+1} \leftarrow c$.
4. Return($(w_{n+1} w_n \cdots w_1 w_0)$).

Za optimalno implementacijo na določenem procesorju je potrebno, da je operacija 2.1 lahko izvedena s pomočjo strojne opreme.

Algoritem odštevanja:

INPUT: positive integers x and y, each having $n + 1$ base b digits, with $x \geq y$.
OUTPUT: the difference $x - y = (w_n w_{n-1} \cdots w_1 w_0)_b$ in radix b representation.

1. $c \leftarrow 0$.
2. For i from 0 to n do the following:
   2.1 $w_i \leftarrow (x_i - y_i + c) \bmod b$.
   2.2 If $(x_i - y_i + c) \geq 0$ then $c \leftarrow 0$; otherwise $c \leftarrow -1$.
3. Return($(w_n w_{n-1} \cdots w_1 w_0)$).

2.1.2 Množenje

Če množimo dve števili x in y, bo imel zmnožek največ toliko cifer, kot jo vsota cifer x in y.

Algoritem množenja:

INPUT: positive integers x and y having $n+1$ and $t+1$ base b digits, respectively.
OUTPUT: the product $x \cdot y = (w_{n+t+1} \cdots w_1 w_0)_b$ in radix b representation.

1. For i from 0 to $(n+t+1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to t do the following:
   - 2.1 $c \leftarrow 0$.
   - 2.2 For j from 0 to n do the following:
     Compute $(uv)_b = w_{i+j} + x_j \cdot y_i + c$, and set $w_{i+j} \leftarrow v$, $c \leftarrow u$.
   - 2.3 $w_{i+n+1} \leftarrow u$.
3. Return($(w_{n+t+1} \cdots w_1 w_0)$).

Ta algoritem deluje enako kot ročna metoda s papirjem in svinčnikom. Algoritem potrebuje (n + 1)(t + 1) običajnih množenj. Predvideva se, da procesor zna izvajati običajna množenja. Računsko najbolj zahteven del algoritma je korak 2.2. $(uv)_b$ se imenuje notranji produkt.

2.1.3 Kvadriranje

INPUT: positive integer $x = (x_{t-1} x_{t-2} \cdots x_1 x_0)_b$.
OUTPUT: $x \cdot x = x^2$ in radix b representation.

1. For i from 0 to $(2t-1)$ do: $w_i \leftarrow 0$.
2. For i from 0 to $(t-1)$ do the following:
   - 2.1 $(uv)_b \leftarrow w_{2i} + x_i \cdot x_i$, $w_{2i} \leftarrow v$, $c \leftarrow u$.
   - 2.2 For j from $(i+1)$ to $(t-1)$ do the following:
     $(uv)_b \leftarrow w_{i+j} + 2x_j \cdot x_i + c$, $w_{i+j} \leftarrow v$, $c \leftarrow u$.
   - 2.3 $w_{i+t} \leftarrow u$.
3. Return($(w_{2t-1} w_{2t-2} \ldots w_1 w_0)_b$).

Računsko zahteven del algoritma je korak 2.2. Število običajnih množenj je približno $(t^2 + t)/2$

2.1.4 Deljenje

Deljenje je najbolj komplicirana in potratna operacija pri dolgih številih.

INPUT: positive integers $x = (x_n \cdots x_1 x_0)_b$, $y = (y_t \cdots y_1 y_0)_b$ with $n \geq t \geq 1$, $y_t \neq 0$.
OUTPUT: the quotient $q = (q_{n-t} \cdots q_1 q_0)_b$ and remainder $r = (r_t \cdots r_1 r_0)_b$ such that $x = qy + r$, $0 \leq r < y$.

1. For j from 0 to $(n - t)$ do: $q_j \leftarrow 0$.
2. While $(x \geq yb^{n-t})$ do the following: $q_{n-t} \leftarrow q_{n-t} + 1$, $x \leftarrow x - yb^{n-t}$.
3. For i from n down to $(t + 1)$ do the following:
   - 3.1 If $x_i = y_t$ then set $q_{i-t-1} \leftarrow b - 1$; otherwise set $q_{i-t-1} \leftarrow \lfloor (x_i b + x_{i-1}) / y_t \rfloor$.
   - 3.2 While $(q_{i-t-1}(y_t b + y_{t-1}) > x_i b^2 + x_{i-1} b + x_{i-2})$ do: $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
   - 3.3 $x \leftarrow x - q_{i-t-1} y b^{i-t-1}$.
   - 3.4 If $x < 0$ then set $x \leftarrow x + yb^{i-t-1}$ and $q_{i-t-1} \leftarrow q_{i-t-1} - 1$.
4. $r \leftarrow x$.
5. Return(q,r).

Ta algoritem potrebuje (n – t)(t + 3) običajnih množenj in do (n – t) običajnih deljenj, ki jih mora procesor podpirati.

2.2 Aritmetika modulov za velika cela števila

Enako kot za velika cela števila je možno zgoraj opisane operacije uporabiti tudi za Z_m, modulo m za cela števila. Naj bo m pozitivno celo število ter x in y ne-negativni celi števili.

V tem poglavju si bomo ogledali algoritme za:

- modularno seštevanje (x + y mod m)
- modularno odštevanje (x – y mod m)
- modularno množenje (x * y mod m)
- modularna inverzija (x^{-1} mod m)

Če je z celo število, potem je z mod m (celoštevilčni ostanek v intervalu [0, m-1] ko je z deljen z m) modularno zmanjšanje z za modulus m.

2.2.1 Modularno seštevanje in odštevanje

Algoritem za seštevanje ostaja enak kot za velika cela števila, le da je potrebno v primeru, ko x + y ≥ m, od rezultata odšteti še m.

Algoritem odštevanja je enak kot za velika cela števila.

2.2.2 Klasično modularno množenje

Klasično modularno množenje je izvedeno s pomočjo množenja velikih celih števil opisanega v prejšnjem poglavju. Pri tem je potrebno po množenju vmesni rezultat deliti s številom m in ostanek pri tem deljenju predstavlja rezultat operacije.

Operacijo lahko zapišemo kot: x * y mod m

2.2.3 Algoritem: Montgomery-jevo množenje

Ta algoritem omogoča bolj učinkovito implementacijo modularnega množenja, saj ni več potreben klasičen izračun ostanka pri deljenju, ki je procesorsko zelo draga operacija.

Za razumevanje je potreben algoritem Montgomery-evega zmanjšanja, ki ne potrebuje več klasičnega algoritma zmanjšanja z uporabo deljenja in ostanka.

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$ with $\gcd(m, b) = 1$, $R = b^n$, $m' = -m^{-1} \bmod b$, and $T = (t_{2n-1} \cdots t_1 t_0)_b < mR$.
OUTPUT: $TR^{-1} \bmod m$.

1. $A \leftarrow T$. (Notation: $A = (a_{2n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n-1)$ do the following:
   2.1 $u_i \leftarrow a_i m' \bmod b$.
   2.2 $A \leftarrow A + u_i m b^i$.
3. $A \leftarrow A / b^n$.
4. If $A \geq m$ then $A \leftarrow A - m$.
5. Return(A).

Potrebno je tudi vedeti, da ta algoritem ne deluje popolnoma za vse vrednosti R, vendar za praktične vrednosti, ki se uporabljajo pri realnih algoritmih (npr. RSA) algoritem zadošča.

Algoritem potrebuje n(n + 1) običajnih množenj.

Sam algoritem Montgomery-evega množenja izgleda takole:

INPUT: integers $m = (m_{n-1} \cdots m_1 m_0)_b$, $x = (x_{n-1} \cdots x_1 x_0)_b$, $y = (y_{n-1} \cdots y_1 y_0)_b$ with $0 \leq x, y < m$, $R = b^n$ with $\gcd(m, b) = 1$, and $m' = -m^{-1} \bmod b$.
OUTPUT: $xyR^{-1} \bmod m$.

1. $A \leftarrow 0$. (Notation: $A = (a_n a_{n-1} \cdots a_1 a_0)_b$.)
2. For i from 0 to $(n-1)$ do the following:
   2.1 $u_i \leftarrow (a_0 + x_i y_0) m' \bmod b$.
   2.2 $A \leftarrow (A + x_i y + u_i m) / b$.
3. If $A \geq m$ then $A \leftarrow A - m$.
4. Return(A).

Algoritem združuje Montgomery-evo zmanjšanje in klasično množenje dveh velikih celih števil.

Število potrebnih enostavnih množenj je 2n(n + 1). Montgomery-evo množenje je zelo učinkovito pri modularnem eksponiranju.

2.2.4 Barret-ovo zmanjšanje

Ta algoritem izračuna vrednost r = x mod m.

INPUT: positive integers $x = (x_{2k-1} \cdots x_1 x_0)_b$, $m = (m_{k-1} \cdots m_1 m_0)_b$ (with $m_{k-1} \neq 0$), and $\mu = \lfloor b^{2k}/m \rfloor$.
OUTPUT: $r = x \bmod m$.

1. $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$, $q_2 \leftarrow q_1 \cdot \mu$, $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$.
2. $r_1 \leftarrow x \bmod b^{k+1}$, $r_2 \leftarrow q_3 \cdot m \bmod b^{k+1}$, $r \leftarrow r_1 - r_2$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r \geq m$ do: $r \leftarrow r - m$.
5. Return(r).

Čeprav algoritem uporablja deljenja, so to zaradi dvojiškega sistema le pomikanja v desno. Take ukaze poznajo vsi sodobni procesorji, zato ta deljenja procesorsko niso zahtevna. Skupno število enostavnih množenj je $(k^2 + 5k + 2)/2 + \binom{k+1}{2} + k$.

2.3 Algoritmi za iskanje največjega skupnega deljitelja

Pogosto se v šifrirnih postopkih uporabljajo algoritmi za iskanje največjega skupnega deljitelja dveh celih števil (gcd). Klasični algoritem potrebuje deljenje velikih celih števil, kar je zelo draga operacija, zato si bomo tukaj ogledali nekaj učinkovitejših algoritmov.

2.3.1 Binarni gcd algoritem

Je dokaj enostaven algoritem, predvsem zato, ker se vsa deljenja izvajajo z deljiteljem 2, kar pri dvojiških računalnikih pomeni premikanje v desno za eno mesto.

INPUT: two positive integers x and y with $x \geq y$.
OUTPUT: $\gcd(x, y)$.

1. $g \leftarrow 1$.
2. While both x and y are even do the following: $x \leftarrow x/2$, $y \leftarrow y/2$, $g \leftarrow 2g$.
3. While $x \neq 0$ do the following:
   3.1 While x is even do: $x \leftarrow x/2$.
   3.2 While y is even do: $y \leftarrow y/2$.
   3.3 $t \leftarrow |x - y|/2$.
   3.4 If $x \geq y$ then $x \leftarrow t$; otherwise, $y \leftarrow t$.
4. Return$(g \cdot y)$.

2.3.2 Lehmerjev gcd algoritem

Ta algoritem je izpeljanka klasičnega Evklidovega algoritma in je primeren za računanje z velikimi celimi števili. V primerjavi s klasično izvedbo uporablja namesto operacij z velikimi števili enostavne operacije.

INPUT: two positive integers x and y in radix b representation, with $x \geq y$.
OUTPUT: $\gcd(x, y)$.

1. While $y \geq b$ do the following:
   1.1 Set $\widetilde{x}$, $\widetilde{y}$ to be the high-order digit of x, y, respectively ($\widetilde{y}$ could be 0).
   1.2 $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$.
   1.3 While $(\widetilde{y} + C) \neq 0$ and $(\widetilde{y} + D) \neq 0$ do the following:
       $q \leftarrow \lfloor (\widetilde{x} + A)/(\widetilde{y} + C) \rfloor$, $q' \leftarrow \lfloor (\widetilde{x} + B)/(\widetilde{y} + D) \rfloor$.
       If $q \neq q'$ then go to step 1.4.
       $t \leftarrow A - qC$, $A \leftarrow C$, $C \leftarrow t$, $t \leftarrow B - qD$, $B \leftarrow D$, $D \leftarrow t$.
       $t \leftarrow \widetilde{x} - q\widetilde{y}$, $\widetilde{x} \leftarrow \widetilde{y}$, $\widetilde{y} \leftarrow t$.
   1.4 If $B = 0$, then $T \leftarrow x \bmod y$, $x \leftarrow y$, $y \leftarrow T$;
       otherwise, $T \leftarrow Ax + By$, $u \leftarrow Cx + Dy$, $x \leftarrow T$, $y \leftarrow u$.
2. Compute $v = \gcd(x, y)$ using Algorithm 2.104.
3. Return(v).

2.3.3 Dvojiško razširjeni gcd algoritem

Klasični algoritem za računanje največjega skupnega deljitelja temelji na enačbi
ax + by = v, kjer v = gcd(x,y). Njegova pomanjkljivost je v tem, da potrebuje drage operacije deljenja velikih števil, kjer sta x in y veliki celi števili. Spodnji algoritem se tej zahtevi izogne s povečanjem števila iteracij izvajanja.

INPUT: two positive integers x and y.
OUTPUT: integers a, b, and v such that $ax + by = v$, where $v = \gcd(x, y)$.

1. $g \leftarrow 1$.
2. While x and y are both even, do the following: $x \leftarrow x/2$, $y \leftarrow y/2$, $g \leftarrow 2g$.
3. $u \leftarrow x$, $v \leftarrow y$, $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$.
4. While u is even do the following:
   4.1 $u \leftarrow u/2$.
   4.2 If $A \equiv B \equiv 0 \pmod 2$ then $A \leftarrow A/2$, $B \leftarrow B/2$; otherwise, $A \leftarrow (A + y)/2$, $B \leftarrow (B - x)/2$.
5. While v is even do the following:
   5.1 $v \leftarrow v/2$.
   5.2 If $C \equiv D \equiv 0 \pmod 2$ then $C \leftarrow C/2$, $D \leftarrow D/2$; otherwise, $C \leftarrow (C + y)/2$, $D \leftarrow (D - x)/2$.
6. If $u \geq v$ then $u \leftarrow u - v$, $A \leftarrow A - C$, $B \leftarrow B - D$;
   otherwise, $v \leftarrow v - u$, $C \leftarrow C - A$, $D \leftarrow D - B$.
7. If $u = 0$, then $a \leftarrow C$, $b \leftarrow D$, and return$(a, b, g \cdot v)$; otherwise, go to step 4.

Ker sta edini potrebni operaciji za delo z velikimi števili seštevanje in odštevanje je algoritem manj procesno zahteven. Deljenje velikih števil poteka vedno z vrednostjo 2, kar pomeni enostavno premikanje v desno za 1 bit. Algoritem potrebuje največ 2(log(x)+log(y)+2) iteracij.

2.4 Eksponiranje

Eksponiranje je ena osnovnih matematičnih operacij za šifrirne postopke z javnim ključem. Uporabljajo jo RSA, Diffie-Hellman, ElGamal in ostali postopki. Prav tako kot za ostale operacije pogosto eksponiranje izvajamo kot modularno operacijo, zato lahko govorimo tako o zapisu g^e kot g^e *mod m*. Operacijo lahko izvajamo nad enim samim elementom ali nad ciklično skupino elementov, ki jo zapišemo kot G.

Najbolj enostavno vendar hkrati tudi najmanj učinkovito bi bilo množiti element g končne ciklične skupine G e – 1 krat. V dobrih šifrirnih postopkih se uporabljajo zelo velike skupine G tako, da je v njih tipično od 2^{160} do 2^{1024} elementov. Z zaporednimi množenji elementov med seboj bi bilo praktično nemogoče izvajati takšne postopke.

Postopek lahko optimiziramo z upoštevanjem:
- zmanjšamo čas potreben za množenje dveh elementov med seboj,
- zmanjšamo število potrebnih množenj za izračun g^e.

Smiselno se je držati obeh načel. Za lažjo implementacijo algoritmov, problem razdelimo na tri podprobleme:

- univerzalne tehnike eksponiranja (dovoljene so vse vrednosti osnove in eksponenta)
- eksponiranje s fiksnim eksponentom (spremenljiva je samo osnova). Take algoritme lahko uporabimo pri RSA.
- eksponiranje s fiksno osnovo (spremenljiv je sam eksponent). Šifriranje in podpisovanje po postopku ElGamal in Diffie-Helman uporabljajo tako vrsto eksponiranja.

2.4.1 Univerzalne tehnike eksponiranja

V tem delu si bomo ogledali algoritme za splošne operacije eksponiranja, kjer sta tako osnova kot eksponent poljubna. Imenujejo se tudi ponovljivi algoritmi kvadriranje-množenje.

Algoritem 1: dvojiško desno-proti-levi eksponiranje

INPUT: an element $g \in G$ and integer $e \geq 1$.
OUTPUT: g^e.

1. $A \leftarrow 1$, $S \leftarrow g$.
2. While $e \neq 0$ do the following:
   2.1 If e is odd then $A \leftarrow A \cdot S$.
   2.2 $e \leftarrow \lfloor e/2 \rfloor$.
   2.3 If $e \neq 0$ then $S \leftarrow S \cdot S$.
3. Return(A).

Če je e naključno izbran kot $0 \leq e < |G| = n$, potem velja, da algoritem potrebuje 3/2 log(n) množenj.

Algoritem 2: dvojiško levo-proti-desno eksponiranje

INPUT: $g \in G$ and a positive integer $e = (e_t e_{t-1} \cdots e_1 e_0)_2$.
OUTPUT: g^e.

1. $A \leftarrow 1$.
2. For i from t down to 0 do the following:
   2.1 $A \leftarrow A \cdot A$.
   2.2 If $e_i = 1$, then $A \leftarrow A \cdot g$.
3. Return(A).

Število množenj je enako kot v prejšnjem algoritmu, razlika je v tem, da je množenje vedno z fiksno vrednostjo g. To lahko znatno pomaga pri mikroprocesorskih implementacijah, kjer lahko vrednost g hranimo v akumulatorjih.

Algoritem 3: metoda eksponiranja z oknenjem

Ta metoda je posplošen algoritem iz prejšnje točke, kjer v vsaki iteraciji algoritem procesira več kot en bit eksponenta.

INPUT: g and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: g^e.

1. *Precomputation.*
   1.1 $g_0 \leftarrow 1$.
   1.2 For i from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
2. $A \leftarrow 1$.
3. For i from t down to 0 do the following:
   3.1 $A \leftarrow A^{2^k}$.
   3.2 $A \leftarrow A \cdot g_{e_i}$.
4. Return(A).

Algoritem 4: metoda z drsečim oknom

Ta algoritem zmanjša obseg pred-procesiranja v primerjavi s prejšnjim algoritmom, zmanjša tudi število potrebnih množenj.

INPUT: g, $e = (e_t e_{t-1} \cdots e_1 e_0)_2$ with $e_t = 1$, and an integer $k \geq 1$.
OUTPUT: g^e.

1. *Precomputation.*
   1.1 $g_1 \leftarrow g$, $g_2 \leftarrow g^2$.
   1.2 For i from 1 to $(2^{k-1} - 1)$ do: $g_{2i+1} \leftarrow g_{2i-1} \cdot g_2$.
2. $A \leftarrow 1$, $i \leftarrow t$.
3. While $i \geq 0$ do the following:
   3.1 If $e_i = 0$ then do: $A \leftarrow A^2$, $i \leftarrow i - 1$.
   3.2 Otherwise ($e_i \neq 0$), find the longest bitstring $e_i e_{i-1} \cdots e_l$ such that $i - l + 1 \leq k$ and $e_l = 1$, and do the following:
   $$A \leftarrow A^{2^{i-l+1}} \cdot g_{(e_i e_{i-1} \ldots e_l)_2}, \quad i \leftarrow l - 1.$$
4. Return(A).

Primerjava vseh štirih algoritmov glede na zahtevnost procesiranja:

Algoritem	Predprocesiranje		Št. kvadriranj	Št. množenj	
	kvadr.	množenj		Najslabši primer	Najboljši primer
1	0	0	t	t	t/2
2	0	0	t	t	t/2
3	1	$2^k - 3$	$t-(k-1) \le lk \le t$	l-1	$l(2^k-1)/2^k$

2.4.2 Eksponiranje s fiksnim eksponentom

Kot že omenjeno, se v šifrirnih postopkih pogosto pojavljajo situacije, ko je potrebno izvesti eksponiranje s fiksnim eksponentom. Primer take uporabe je RSA šifriranje in odšifriranje in ElGamal odšifriranje. To dejstvo naslednji algoritmi uporabljajo za optimizacijo delovanja tako, da potrebujejo manjše število množenj.

Algoritem 1: Eksponiranje z verižnim seštevanjem

Pri tem algoritmu je s dolžina verige, ki je sestavljena iz pozitivnih celih števil 1 do e.

INPUT: a group element g, an addition chain $V = (u_0, u_1, \ldots, u_s)$ of length s for a positive integer e, and the associated sequence $w_1, \ldots, w_s$, where $w_i = (i_1, i_2)$.
OUTPUT: g^e.

1. $g_0 \leftarrow g$.
2. For i from 1 to s do: $g_i \leftarrow g_{i_1} \cdot g_{i_2}$.
3. Return(g_s).

Algoritem potrebuje s množenj za izračun eksponenta.

Algoritem 2: Veriga s seštevanjem vektorjev

V primerjavi s prejšnjim algoritmom ta algoritem uporablja verige vektorjev za seštevanje.

INPUT: group elements $g_0, g_1, \ldots, g_{k-1}$ and a vector-addition chain V of length s and dimension k with associated sequence $w_1, \ldots, w_s$, where $w_i = (i_1, i_2)$.
OUTPUT: $g_0^{e_0} g_1^{e_1} \cdots g_{k-1}^{e_{k-1}}$ where $v_s = (e_0, e_1, \ldots, e_{k-1})$.

1. For i from $(-k+1)$ to 0 do: $a_i \leftarrow g_{i+k-1}$.
2. For i from 1 to s do: $a_i \leftarrow a_{i_1} \cdot a_{i_2}$.
3. Return(a_s).

2.4.3 Eksponiranje s fiksno osnovo

V tem delu so predstavljene metode za izračun eksponentov, ko je osnova konstanta, eksponent pa se spreminja. To lahko izkoristimo, da predračunanje opravimo samo enkrat, potem pa glede na vrednost eksponenta rezultat samo popravljamo. Ta način na primer uporablja Diffie-Hellman

2.4.4 Metoda z oknenjem

Ta algoritem kot vhod vzame predizračunane vrednosti eksponentov $g_i = g^{b_i}$, $0 \le i \le t$. Te vrednosti je potrebno izračunati z enim izmed gornjih algoritmov. Osnova algoritma je enačba:

$$g^e = \prod_{i=0}^{t} g_i^{e_i} = \prod_{j=1}^{h-1} (\prod_{e_i=j} g_i)^j$$

INPUT: $\{g^{b_0}, g^{b_1}, \ldots, g^{b_t}\}$, $e = \sum_{i=0}^{t} e_i b_i$, and h.
OUTPUT: g^e.

1. $A \leftarrow 1$, $B \leftarrow 1$.
2. For j from $(h-1)$ down to 1 do the following:
   - 2.1 For each i for which $e_i = j$ do: $B \leftarrow B \cdot g^{b_i}$.
   - 2.2 $A \leftarrow A \cdot B$.
3. Return(A).

Ta algoritem potrebuje do t + h – 2 množenj za izračun eksponenta.

2.4.5 Evklidova metoda za fiksno osnovo

INPUT: $\{g^{b_0}, g^{b_1}, \ldots, g^{b_t}\}$ and $e = \sum_{i=0}^{t} e_i b_i$.
OUTPUT: g^e.

1. For i from 0 to t do the following: $g_i \leftarrow g^{b_i}$, $x_i \leftarrow e_i$.
2. Determine the indices M and N for $\{x_0, x_1, \ldots, x_t\}$.
3. While $x_N \neq 0$ do the following:
   - 3.1 $q \leftarrow \lfloor x_M / x_N \rfloor$, $g_N \leftarrow (g_M)^q \cdot g_N$, $x_M \leftarrow x_M \bmod x_N$.
   - 3.2 Determine the indices M and N for $\{x_0, x_1, \ldots, x_t\}$.
4. Return($g_M^{x_M}$).

Procesno je ta algoritem podoben prejšnjemu, njegova glavna značilnost je, da potrebuje manj prostora za shranjevanje podatkov.

2.4.6 „Comb“ metoda za fiksno osnovo

Za ta algoritem je potrebno opraviti predprocesiranje po sledečem algoritmu:

INPUT: group element g and parameters h, v, a, and b (defined above).
OUTPUT: $\{G[j][i] : 1 \leq i < 2^h, 0 \leq j < v\}$.

1. For i from 0 to $(h-1)$ do: $g_i \leftarrow g^{2^{ia}}$.
2. For i from 1 to $(2^h - 1)$ (where $i = (i_{h-1} \cdots i_0)_2$), do the following:
   2.1 $G[0][i] \leftarrow \prod_{j=0}^{h-1} g_j^{i_j}$.
   2.2 For j from 1 to $(v-1)$ do: $G[j][i] \leftarrow (G[0][i])^{2^{jb}}$.
3. Return($\{G[j][i] : 1 \leq i < 2^h, 0 \leq j < v\}$).

Sam algoritem pa izgleda takole:

INPUT: g, e and $\{G[j][i] : 1 \leq i < 2^h, 0 \leq j < v\}$ (precomputed in Algorithm 14.116).
OUTPUT: g^e.

1. $A \leftarrow 1$.
2. For k from $(b-1)$ down to 0 do the following:
   2.1 $A \leftarrow A \cdot A$.
   2.2 For j from $(v-1)$ down to 0 do: $A \leftarrow G[j][I_{j,k}] \cdot A$.
3. Return(A).

Algoritem potrebuje največ a + b – 2 množenj, kjer je a = (t+1)/h in b=a/v.

2.5 Prekodiranje eksponenta

Drug način za zmanjšanje števila množenj za izračun eksponenta je spreminjanje osnove zapisa eksponenta e na tak način, da je uporabljeno manj vrednosti različnih od nič. Ker je digitalni zapis unikaten (1 in 0) je potrebno uvesti zapise, kjer ima ena števka več vrednosti. To lahko naredimo na več načinov. Tukaj si bomo ogledali tehniki z uporabo:

- predznaka za vsako števko,
- menjave nizov.

2.5.1 Zapis z uporabo predznaka

Če eksponent zapišemo kot:

$e = \sum_{i=0}^{t} d_i 2^i$ where $d_i \in \{0, 1, -1\}$, $0 \leq i \leq t$, then $(d_t \cdots d_1 d_0)_{SD}$

potem govorimo o zapisu z uporabo predznaka.

Naslednji algoritem nam spremeni eksponent zapisan v dvojiški obliki dolžine t+1 v eksponent zapisan v obliki s predznakom, katere največja dolžina je t+1, vendar uporablja največje možno število ničelnih vredností.

INPUT: a positive integer $e = (e_{t+1}e_te_{t-1} \cdots e_1e_0)_2$ with $e_{t+1} = e_t = 0$.
OUTPUT: a signed-digit representation $(d_t \cdots d_1d_0)_{SD}$ for e. (See Definition 14.120.)

1. $c_0 \leftarrow 0$.
2. For i from 0 to t do the following:
   2.1 $c_{i+1} \leftarrow \lfloor (e_i + e_{i+1} + c_i)/2 \rfloor$, $d_i \leftarrow e_i + c_i - 2c_{i+1}$.
3. Return($(d_t \cdots d_1d_0)_{SD}$).

Izvajanje tega algoritma je lahko zelo učinkovito, če vnaprej izračunamo vrednosti in jih zapišemo v tabelo, nato pa jih ob uporabi samo iščemo po tabeli in jih ne izračunavamo več sproti.

2.5.2 Zapis z menjavanjem nizov

INPUT: $e = (e_{t-1}e_{t-2} \cdots e_1e_0)_2$ and positive integer $k \geq 2$.
OUTPUT: $e = (f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$.

1. For i from k down to 2 do the following: starting with the most significant digit of $e = (e_{t-1}e_{t-2} \cdots e_1e_0)_2$, replace each consecutive string of i ones with a string of length i consisting of $i - 1$ zeros in the high-order string positions and the integer $2^i - 1$ in the low-order position.
2. Return($(f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$).

Če vzamemo za primer $e=110111110011101_2$ in k=3, bo algoritem kot rezultat dal $e=030007030000701_{SR(3)}$.

Celoten algoritem za eksponiranje izgleda takole:

INPUT: an integer $k \geq 2$, an element $g \in G$, and $e = (f_{t-1}f_{t-2} \cdots f_1f_0)_{SR(k)}$.
OUTPUT: g^e.

1. *Precomputation*. Set $g_1 \leftarrow g$. For i from 2 to k do: $g_{2^i-1} \leftarrow (g_{2^{i-1}-1})^2 \cdot g$.
2. $A \leftarrow 1$.
3. For i from $(t-1)$ down to 0 do the following:
   3.1 $A \leftarrow A \cdot A$.
   3.2 If $f_i \neq 0$ then $A \leftarrow A \cdot g_{f_i}$.
4. Return(A).

Predračunanje pri tem algoritmu potrebuje k-1 kvadriranj in k-1 množenj. Pričakovati je, da se za korak 3 porabi v praksi t-1 kvadriranj in t/4 množenj.

3 ALTIVEC

V prejšnjem poglavju smo si ogledali algoritme za procesorske sisteme na splošno, v nadaljevanju pa si oglejmo, kako lahko procesiranje še dodatno pohitrimo z uporabo pospeševalnega bloka za vektorsko računanje AltiVec.

AltiVec je bil razvit kot dodatek procesorjev PowerPC. Skupaj so ga razvili Apple, Motorola in IBM. Vsako podjetje zanj uporablja svoje ime, zato lahko naletimo na:

- Velocity Engine (Apple)
- AltiVec (Motorola, sedaj Freescale)
- VMX (IBM)

AltiVec je bil prvi SIMD sistem, ki se je pojavil v namiznih računalnikih (v poznih devetdesetih letih). Kljub temu, da je že dokaj star še vedno prekaša vse najnovejše tekmece podjetij Intel in AMD, saj omogoča uporabo večjih vektorjev in boljšega paralelizma, ter več ukazov. AltiVec se je začel uporabljati v računalnikih Apple Macintosh s procesorji G4, danes pa ga najdemo v zelo širokem krogu naprav. Omogoča učinkovito uporabo pri:

- stiskanju govora pri VoIP
- razpoznavanje govora
- procesiranje zvoka
- procesiranje komunikacijskih kanalov
- 2D in 3D igre, pomoč pri izračunavanju
- procesiranje slik in videa: JPEG, filtri
- izničevanje odmeva pri govornih zvezah pri velikih dolžinah repa (500 ms)
- procesiranje številskih polj
- …

Ne glede na to, ali je AltiVec enota vključena v 32 ali 64-bitno procesorsko jedro, uporablja 128-bitne registre. V en tak register lahko spravimo 16 8-bitnih, 8 16-bitnih ali 4 32-bitnih celoštevilčnih vrednosti ali vrednosti s plavajočo vejico. AltiVec zna tudi pazljivo delati s predpomnilnikom za učinkovito procesiranje podatkovnih tokov.

Podobno, vendar manj učinkovito, tehnologijo poskuša vpeljati tudi Intel s svojimi dodatki MMX, SSE, SSE2, vendar AltiVec premore 32 128-bitnih registrov, medtem ko SSE2 premore le 8 takih registrov. Naslednja prednost AltiVec tehnologije je tudi v tem, da večina ukazov hkrati izvaja operacije na 3 registrih, medtem kot SSE2 uporablja le 2 registra hkrati. Popolnoma unikatni so tudi ukazi za permutacije, kjer lahko podatke iz enega registra prenašamo v drugega s tem da se parametri tega prenosa nahajajo v tretjem registru. Tako lahko samo z enim ukazom počnemo neverjetne stvari.

Po zmogljivosti velja, da AltiVec lahko v enem urinem ciklu izvede 8 32-bitnih operacij s plavajočo vejico, ko jih lahko SSE2 v istem času izvede le 4. Da bi dosegli enako število FLOP-ov, bi morali x86 procesorji delovati z dvojno frekvenco glede na PowerPC procesorje z AltiVecom, kar pa ni mogoče, zato je za določene multimedijske in računske operacije PowerPC kljub nižji frekvenci hitrejši kot x86.

3.1 Princip delovanja AltiVec

Razlika med enoto AltiVec in skalarnimi celoštevilčnimi enotami (integer units) in enotami za delo s plavajočo vejico (FPU, Floating Point Unit) je v tem, da skalarne enote z enim ukazom obdelajo le en podatek, AltiVec pa paralelno obdela več podatkov hkrati. Tako procesiranje imenujemo SIMD (Single Instruction Multiple Data).

Razlika je najlepše vidna pri enostavnem seštevanju. Pri celoštevilčni enoti bi zapisali 1+1=2, pri FPU enoti 1,0+1,0=2,0.

Vendar pri vektorski enoti zapišemo enako operacijo takole:

```
//prištej vector1 k vector2, rezultat shrani v resultVector
resultVector = vec_add(!vector1, vector2).
```

Bistvena razlika je sedaj v tem, da smo pri skalarni enoti sešteli le dve številki med seboj, pri vektorski pa 32, ker le-ta uporablja 128-bitne registre, kjer vsak vsebuje 16 števil. Grafično ponazorjeno poteka seštevanje takole:

vector 1:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
								+								
vector 2:	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
								=								
result vector:	2	3	4	5	7	8	9	10	12	13	14	15	17	18	19	20

Vsak element vektorja 1 je bil prištet k istoležnemu elementu vektorja 2, rezultat pa shranjen na ustrezno mesto tretjega vektorja. Za enako operacijo smo z AltiVecom porabili 16 krat manj časa kot s skalarno enoto. Zaradi tega je lahko kaj hitro jasno, zakaj je AltiVec toliko hitrejši.

3.2 Podatkovni tipi

AltiVec uporablja 32 128-bitnih registrov, v katere je možno shraniti:
- 128 posameznih bitov
- 16 8-bitnih znakov
- 8 kratkih besed
- 8 16-bitnih RGB pik
- 4 cela števila
- 4 števila s plavajočo vejico po IEEE-754 (single precision)

Celoštevilčni tipi so lahko z ali brez predznaka.

3.3 AltiVec operacije

Nad vektorji lahko izvajamo precej različnih operacij. Razdelimo jih v različne skupine:
- pretvorniki tipov
- nastavljanje konstant
- matematične operacije
- logične operacije
- primerjave
- operacije permutacij
- operacije s spominom

3.4 Pretvorniki tipov

Ko se v pretvarjajo različni tipi med seboj moramo biti pozorni ali je določen tip možno direktno pretvoriti v drugega ali ne. Na primer konstanto vrednosti 0 je enostavno pretvoriti v vse ostale zapise, kar pa ni možno za primer, ko bi radi pretvorili število s plavajočo vejico 14,0 celoštevilski zapis 14. Bitni zapis obeh tipov, čeprav za isto vrednost, ni enak, zato enostavno prirejanje spremenljivk ne pride v poštev. Za take primere AltiVec predvideva posebne ukaze, ki so prikazani v spodnji tabeli.

From\To:	char	short	int	16-bit pixel	32-bit pixel	float
char	-	vec_unpackh, vec_unpackl	Convert to short first	X	X	Convert to int first
short	vec_pack, vec_packs, vec_packsu	-	vec_unpackh, vec_unpackl	Static typecast	X	Convert to int first
int	Convert to short first	vec_pack, vec_packs, vec_packsu	-	X	Static Typecast	vec_ctf
16-bit pixel	X	Static typecast	X	-	vec_unpackh, vec_unpackl	Convert to 32-bit pixels first
32-bit pixel	X	X	Static typecast	vec_packpx	-	Convert channels to ints
float	Convert to ints first	Convert to ints first	vec_ctu, vec_cts	Convert to 32-bit pixel first	Convert to int first	-

3.5 Nastavljanje konstant

Ob začetku računanja moramo vektorje napolniti s pravimi konstantami. Običajno se to naredi tako, da iz neke lokacije v pomnilniku preberemo podatke v register. Vendar tak način lahko postane zelo neučinkovit v primeru, da se ta podatek trenutno ni nahajal v predpomnilniku. V takem primeru se lahko zgodi, da mora procesna enota počakati 35-250 ciklov, da se podatek prebere iz pomnilnika.

AltiVec ima zato posebne ukaze za generiranje konstant, kjer lahko na primer ustvarimo vektor samih vrednosti 0x01 ali 0x00000001, itd.

Obstajajo tudi ukazi za generacijo padajočih in naraščajočih nizov, ki so zapisani kot vektorji.

3.6 Matematične operacije

Seštevanje in odštevanje

Ti dve osnovni operaciji z vektorji omogočata vzporedno seštevanje in odštevanje. Posebnost pri vektorskih operacijah je preplavitev, saj lahko s tem bistveno spremenimo vrednost rezultata, če ne upoštevamo „carry“ bita. V ta namen obstajajo posebni ukazi, ki v primeru, da bi prišlo do preplavitve, uporabijo največji ali najmanjši možen rezultat glede na to, ali gre za seštevanje ali odštevanje.

Množenje

Enostavno množenje v AltiVecu ni možno, saj se vedno uporablja v obliki:

rezultat = A * B + C

Kadar želimo izvesti enostavno množenje postavimo C na 0.

Deljenje

Deljenje je direktno možno le z uporabo tipov s plavajočo vejico. Deljenje celih števil je možno s pomočjo spodnjega zapisa:

$$\text{Saturate_to_SInt16}\left(\frac{A \times B + 16384}{32768} + C \right)$$

B je recipročna vrednost deljitelja, ki jo dobimo z ukazom vec_re.

Kvadratni koreni in recipročni kvadratni koreni

AltiVec za tipe s plavajočo vejico podpira tudi korenjenje.

3.7 Logične operacije

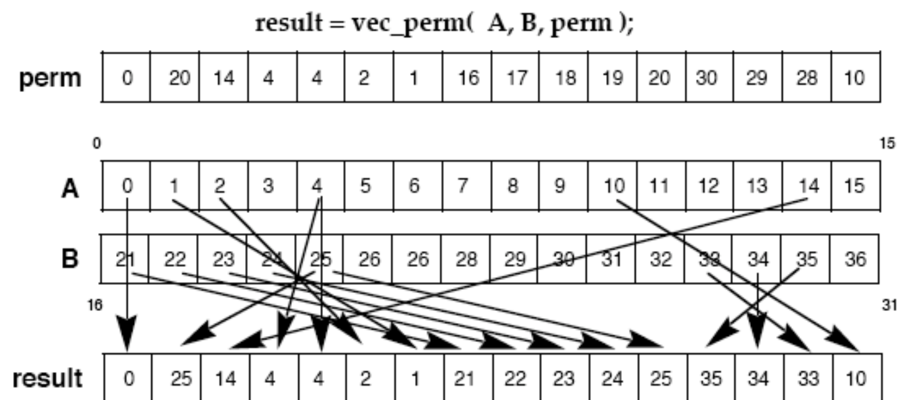
Vse klasične logične operacije kot so and, or, xor, … se lahko izvajajo nad celimi vektorji. To pomeni, da hkrati delamo s 128 biti. Poleg tega obstajajo tudi ukazi za in s komplementom in NOR.

3.8 Primerjave

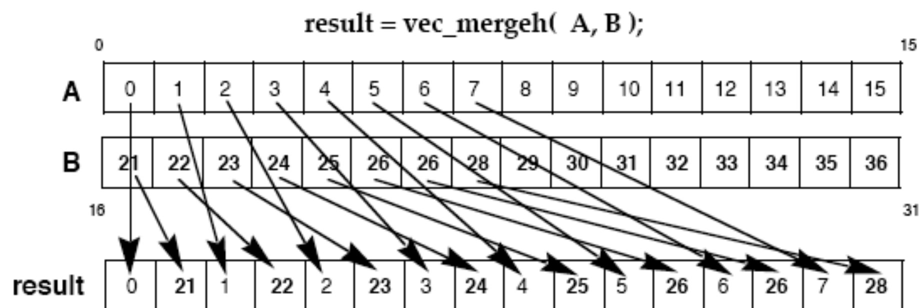
Primerjalne operacije lahko izvajamo za posamezne elemente vektorja, na različne načine (večji, manjši, enak, …). Specifična je funkcij, ki preverja, če določen element vektorja pade v določeni interval ali iz njega.

3.9 Operacije permutacij

Te operacije uporabimo, kadar želimo menjati elemente znotraj vektorja med seboj, lahko pa jih menjamo celo med različnimi vektorji. Zanimiva je operacija, ko v en register damo izvorni vektor, v drugega zapišemo, kako naj se posamezni bajti premešajo med seboj preden se zapišejo v tretji register.



Uporabna je tudi operacija združevanja elementov dveh vektorjev v skupni vektor.



Z eno operacijo je možno narediti tudi premikanje vseh 128 bitov naenkrat v levo ali desno.

3.10 Operacije s spominom

Operacije so podobne tipičnim load/store operacijam, kot jih poznajo običajne procesorske enote, le da je pri AltiVecu potrebno biti zelo pazljiv glede poravnave. Namreč v spominu ne moremo uporabljati kar katerokoli lokacijo za shranjevanje podatkov, saj bi sicer bili mehanizmi predpomnilnika in modernih DRAM-ov zelo slabo izkoriščeni in bi dajali temu primerne rezultate.

4 UPORABA ALTIVEC ZA RSA

V okviru te seminarske naloge nas bolj kot tehnične podrobnosti AltiVec zanimajo predvsem njegove praktične prednosti pri uporabi v šifrirnih postopkih. Čeprav je bila tehnologija AltiVec načrtovana za multimedijske aplikacije je bilo že v sami predstavitvi moč opaziti združljivost z zahtevami šifrirnih postopkov.

Ogledali si bomo SIMD rešitev RSA problemov, ki temelji na AltiVec izvedbi. Čeprav se za RSA smatra, da je zelo sekvenčen problem, ker uporablja modularno eksponiranje z velikimi števili, se bo izkazalo, da paralelizem, ki ga vpelje AltiVec zelo uspešno rešuje probleme.

V kriptografiji z uporabo javnega ključa ima vsakdo par ključev, javnega in privatnega. Privatni ključ je objavljen za vse, ki želijo poslati šifrirano sporočilo prejemniku. Privatni ključ ostaja skrivnost in ga pozna samo prejemnik sporočila. Pri RSA je potrebno na začetku izbrati dve veliki praštevili. Produkt teh dveh števil je javni ključ, ki se ga objavi. Dandanes velja, da je edini primerni način za razbijanje RSA faktorizacija tega produkta. Problem nastopi v tem, da je ta produkt zelo velika številka, 1024 bitno število predstavlja 310 desetiških cifer. Za današnje računalniške sisteme je problem praktično nerešljiv.

4.1 Ozadje RSA

Naj bosta p in q dve veliki praštevili in N njun produkt pq. (N=pq). M je čistopis, ki ga je potrebno šifrirati, rezultat šifriranja pa je C – šifrirano sporočilo. e je javno znan parameter kot del javnega ključa. RSA postopek poteka takole:

Javni ključ: (e,N)
Privatni ključ: (d,N) kjer $d=e^{-1} \bmod (p-1)(q-1)$
Čistopis: M
Šifrirano sporočilo: C

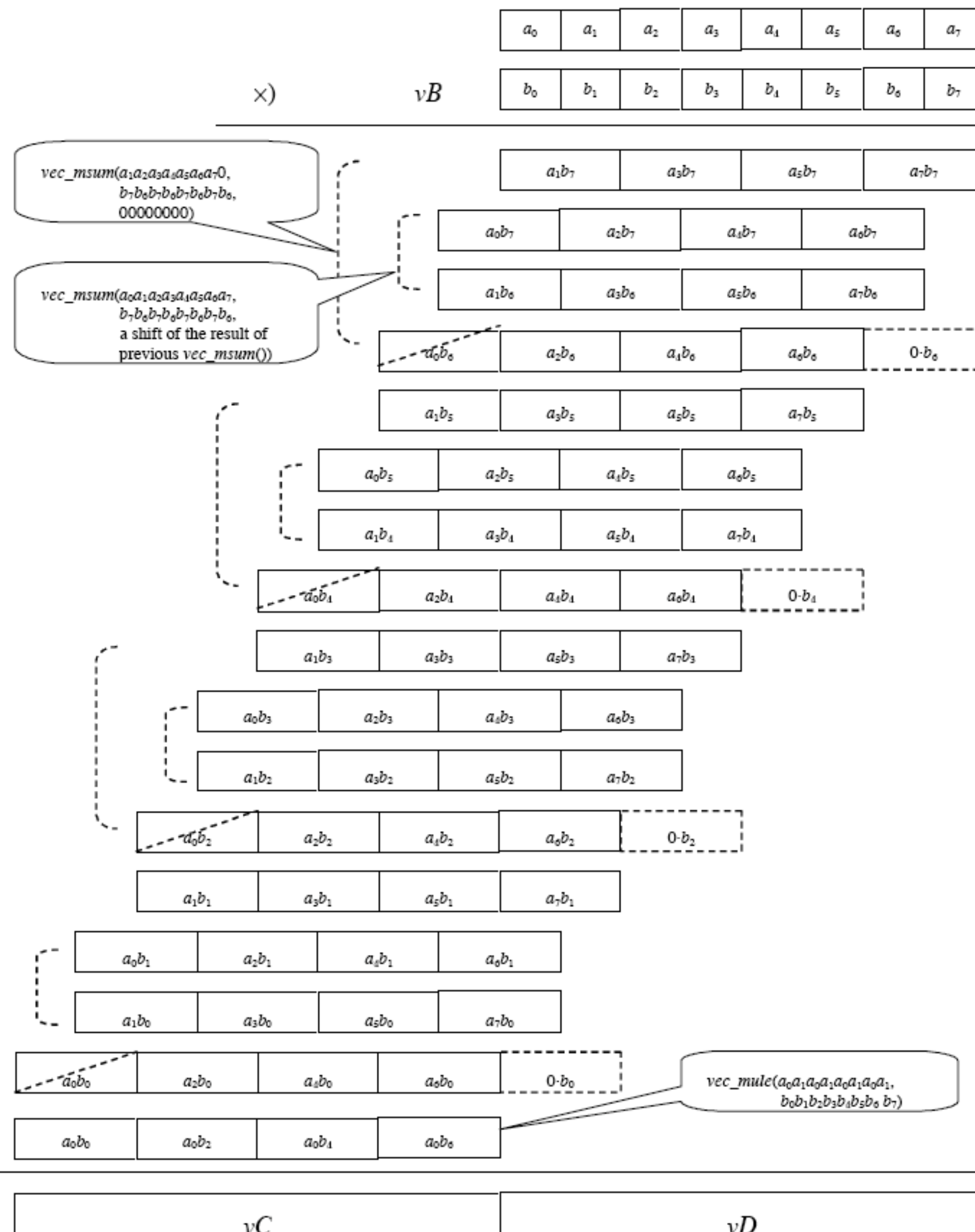
Šifriranje: $C = M^{e} \bmod N$

Dešifriranje: $M = C^{d} \bmod N$

Najlažji način za pridobiti d je faktorizacija N. Oseba nikoli ne sme objaviti (d,N), mora pa objaviti svoj javni ključ (e,N).

4.2 Uporaba AltiVec za modularno eksponiranje

Za učinkovito izvedbo RSA1024 je potrebno najti čim boljši način za izvedbo množenja dveh 128 bitnih števil – 128 bit x 128 bit, rezultat je velikosti 256 bit.



1. Po tem postopku so 16-bitni elementi vektorjev vA in vB s pomočjo AltiVec ukazov za permutacije prerazporejeni v začasne vektorske registre kot začasni vektorji za vec_msum ukaz.

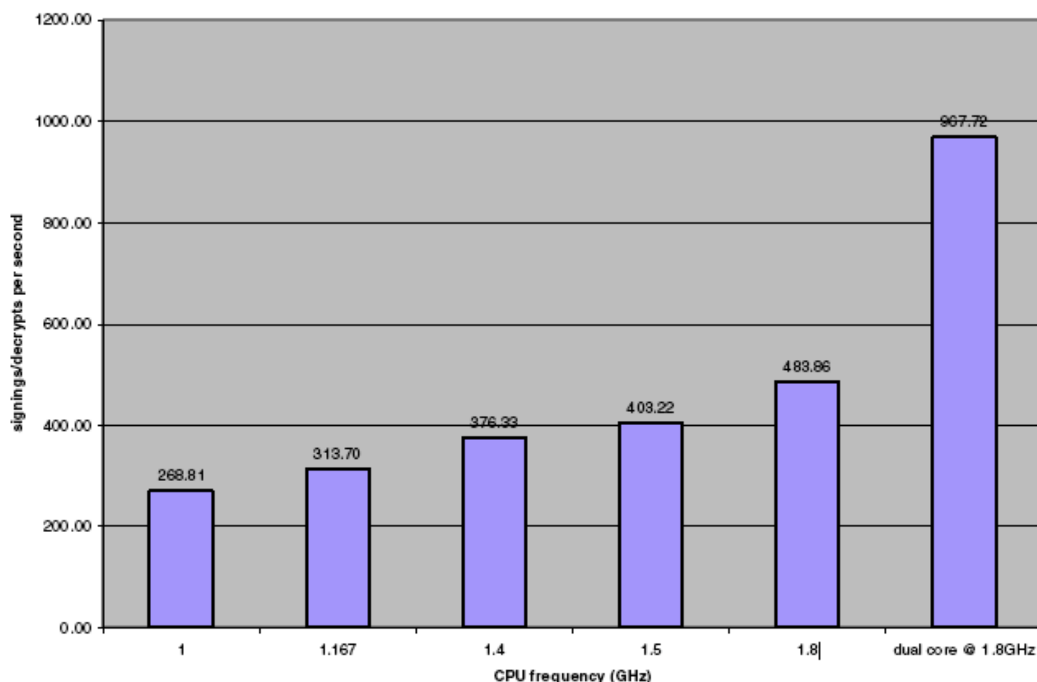
2. Z enim ukazom vec_mule izračunamo štiri produkte a_0b_0, a_0b_2, a_0b_4, a_0b_6, da lahko obdržimo poravnavo vpletenih vektorjev z ukazom vec_msum.
3. Vsota vseh rezultatov vec_msum in rezultata vec_mule je enaka produktu obeh velikih števil vA in vB

Čeprav ta algoritem potrebuje veliko ukazov za izračun zmnožka je izjemno učinkovit zaradi več razlogov:
- AltiVec ukazi so razvrščeni v tri sočasne enote izvajanja (VPU, VIU1 in VIU2), vsaka enota lahko v enem urinem taktu izvede en ukaz
- e600 jedro lahko izvede dva AltiVec ukaza v enem urinem taktu
- AltiVec uporablja 128-bitna nalaganja in shranjevanja podatkov, kar prihrani precej časa

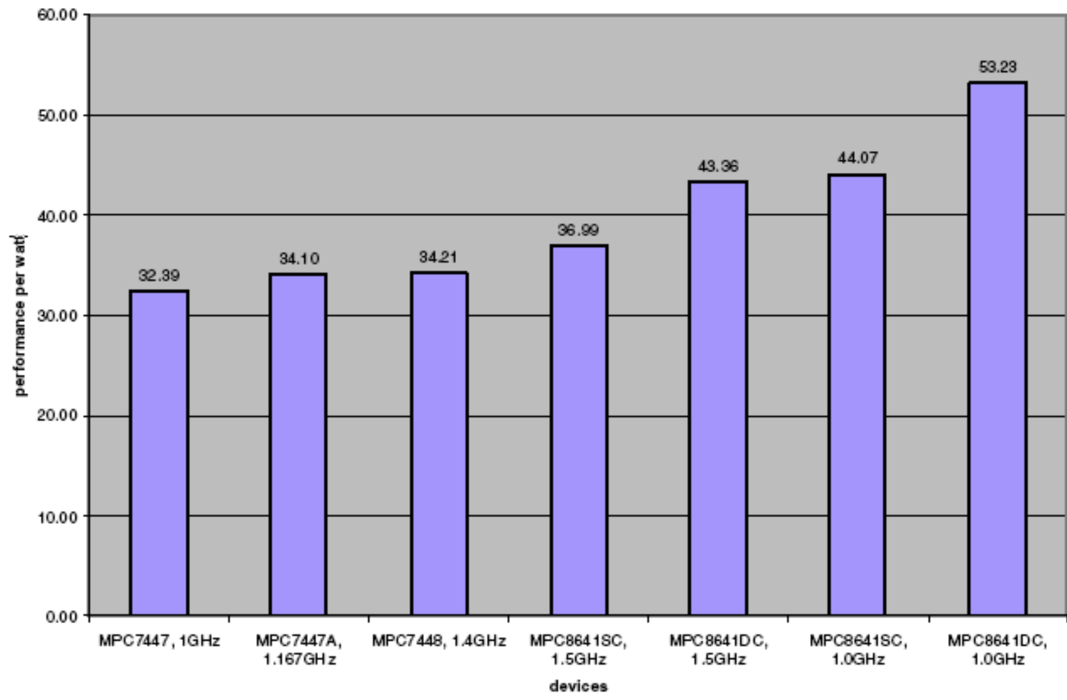
Zato je s pomočjo Montgomeryevega algoritma možno zelo učinkovito izračunati modularno eksponiranje velikih števil.

1024-bitno RSA dešifriranje je zato možno na procesorju e600 MPC7447 hitrosti 1 GHz izvesti v manj kot 3,72 ms kar je enako 268,81 podpisov v eni sekundi.

Spodnja slika prikazuje RSA zmogljivost v odvisnosti od frekvence procesorja. Pri dvo-procesorskem jedru lahko dosežemo dvojno zmogljivost, kar lahko izkoristimo za večje število dešifriranj v eni sekundi ali za krajši čas dešifriranja. To lahko naredimo tako, da razbijemo eno modularno eksponiranje velikih števil na dve eksponiranji pol manjših števil.



Zaključimo lahko, da je AltiVec zelo primerna tehnologija za izvajanje RSA algoritma, sploh v vgrajenih sistemih, kjer je zelo pomembno razmerje procesna zmogljivost/porabljeno električno moč.


60.00
50.00
40.00
30.00
20.00
10.00
0.00
performance per watt
32.39
34.10
34.21
36.99
43.36
44.07
53.23
MPC7447, 1GHz
MPC7447A, 1.167GHz
MPC7448, 1.4GHz
MPC8641SC, 1.5GHz
MPC8641DC, 1.5GHz
MPC8641SC, 1.0GHz
MPC8641DC, 1.0GHz
devices

5 ALTIVEC IN KONČNA POLJA

V tem poglavju si bomo ogledali kako je možno tehnologijo AltiVec uporabiti za pospešitev osnovnih operacij v vektorskih prostorih preko končnih polj. Bolj natančno si bomo pogledali tiste postopke, ki so specifični za TTM (The Tame Transformation Method). Ta metoda temelji na permutacijah.

Polja so množice objektov, na katerih lahko izvajamo operacije seštevanja in množenja. Od polja zahtevamo, da veljajo vse klasične zakonitosti operacij:

- so kumulativne
- so asociativne
- da so imajo množenja večjo težo kot seštevanja
- da imajo identitete
- …

Končna polja označujemo kot Z_n, kjer n pove koliko objektov ima polje. Kadar n=2 govorimo o dvojiškem polju, ki ima samo objekta 0 in 1.

5.1 Seštevanje

Seštevanje je tudi v končnih poljih najenostavnejša operacija. Pri dvojiških poljih je seštevanje kar enako funkciji XOR.

Če imamo kazalca na polji a in b s spodnjo funkcijo enostavno izračunamo vsoto polj. Rezultat je zapisan na naslov kamor kaže kazalec c.

```
void finite_field_vector_addition (unsigned char *a,
                        unsigned char *b,
                        unsigned char *c,
                        int n)
{
      int i;
      vector unsigned char *vec_a = (vector unsigned char *) a;
      vector unsigned char *vec_b = (vector unsigned char *) b;
      vector unsigned char *vec_c = (vector unsigned char *) c;
      for (i = 0; i < n; i += 16)
            *vec_c++ = vec_xor (*vec_a++, *vec_b++);
}
```

Ta funkcija izvaja seštevanje 128 bitov na enkrat.

5.2 Množenje

Skalarno množenje je nekoliko bolj komplicirano, saj želimo izračunati $y=\alpha x$ kjer je α skalar (na primer element polja z 256 vrednostmi) in x vektor n elementov iz istega polja. Predpostavimo vsaj, da je n večkratnik števila 16, da nimamo težav z poravnavami v AltiVecu.

Vsak element y_i zapišemo kot $y_i=\alpha \times x_i$, kjer $0 \leq I \leq n$

Če zapišemo bitni zapis za x_i = abcdefgh vidimo, da

$\alpha \times x_i = \alpha \times \text{abcdefgh} = \alpha \times (\text{0000efgh} + \text{abcd0000}) = \alpha \times \text{0000efgh} + \alpha \times \text{abcd0000}$

x pomeni množenje v končnem polju, + pa seštevanje v istem polju.

Za vseh 256 različnih vrednosti α izračunamo tabelo 16 različnih spodnjih produktov $\alpha \times$ 0000efgh, efgh=0000, 0001, …, 1111 in prav tako tudi tabelo 16 različnih zgornjih produktov $\alpha \times$ abcd0000, abcd=0000, 0001, …, 1111.

V podprogramu za množenje se zavedamo vrednosti α in naložimo pravi dve tabeli v 2 vektorska registra. Vektorja obravnavamo kot vektorja tipa unsigned char. Množenje spodnjega dela x_i z α je enostavno iskanje po tabeli delnih produktov. Enako velja za zgornji del.

Ukaz vec_perm je idealen za iskanje po kratkih tabelah (<16) 8-bitnih vrednosti. V splošnem bi lahko uporabili 16 sočasnih iskanj po 32 bajtni tabeli, 16 kriterijev je zapisanih v enem vektorju, dve 16-bajtni tabeli pa v ostalih dveh. Spodnjih 5 bitov kriterija se uporabi kot indeks na tabelo.

Celoten algoritem izgleda takole:

```
vector unsigned char low_products[256];
vector unsigned char high_products[256];
/* Initialize tables of low products and high products here. */
(void)
finite_field_scalar_multiplication (unsigned char alpha,
                                    unsigned char *x,
                                    unsigned char *y,
                                    int n)
{
      int i;
      vector unsigned char *vec_x = (vector unsigned char *) x;
      vector unsigned char *vec_y = (vector unsigned char *) y;
      vector unsigned char low = low_products[(int) alpha];
```

```
    vector unsigned char high = high_products[(int) alpha];
    for (i = 0; i < n; i += 16, vec_x++, vec_y++)
    {
          vector unsigned char l, h;
          l = vec_perm (low, low, *vec_x);
          h = vec_perm (high,high,
                      vec_sr (*vec_x, (vector unsigned char) (4)));
          *vec_y = vec_xor (l, h);
    }
}
```

Za vsako vrednost i-ja zanka potrebuje 8 ukazov ko je prevedemo v strojni jezik procesorja.

5.3 Zmogljivost

V tem delu bomo primerjali zmogljivost programov za šifriranje po Tame Transformation Metodi (TTM) z uporabo skalarnih in z uporabo AltiVec ukazov. Osredotočimo se na proces šifriranja, ker je le-ta bolj zahteven.

TTM je bločni šifrirnik. Vhodni blok je dolg m bajtov, izhodni n, za naš primer
$n \geq m + 36$

Javni ključ ima dva dela:

- matrični ključ: sestoji iz n spodnjih trikotnih matrik A_i, vsaka velikosti m x m, z elementi končnega polja
- vektorskega ključa iz n vektorjev v_i dolžine m z elementi iz končnega polja.

Če je x vhodno sporočilo. i-ti izhodni bajt izračunamo kot:

$$x^T * (v_i + A_i x)$$

Privatni ključ je sestavljen iz treh delov:

- matrika m x m
- matrika n x n
- vektor dolžine m

Število bitov privatnega ključa je $(m^2 + n^2 + m)$ krat število bitov posameznega vnosa ključa.

Izkaže se, da 8 bitni vnos popolnoma zadošča za varnost postopka. Izkaže se, da je 4 bitni vnos popolnoma zadovoljiv, kar še dodatno poenostavi vektorske operacije, saj so vsi podatki še enkrat ožji. Potrebujemo samo eno nalaganje vektorja, eno vektorsko permutacijo, en vektorski ekskluzivni-ali za 16 skalarnih množenj+seštevanj.

Skalarno napisan program za arhitekturo PowerPC e600 potrebuje za kombinirano operacijo množenja in seštevanja 2,56 ciklov (2 load ukaza in 1 xor).
Iz te ugotovitve lahko sklepamo, da bo skalarna izvedba šifriranja potrebovala

$$2.56 \times \left(\frac{m(m+1)}{2} + m \right) \times n$$

ukazov za šifriranje enega bloka.

V spodnji tabeli so prikazani še rezultati za algoritem, ki uporablja AltiVec, vidimo, da je izboljšanje veliko.

Vhodni/izhodni blok	**28/64 bajtov**	**36/72 bajtov**
Skalarna hitrost	1.260.080 bits/sec	890.313 bits/sec
AltiVec	18.801.311 bits/sec	7.488.117 bits/sec
AltiVec, cache sync		9.841.225 bits/sec

Vidimo, da smo z uporabo AltiVec tehnologije dosegli izboljšanje za faktor od 11 do 14, odvisno od dolžine blokov.

6 VIRI

[1] A. Menezes, P. van Oorschot, S. Vanstone: Handbook of Applied Cryptography. CRC Press. 1996.

[2] IBM: PowerPC Microprocessor Family: AltiVec Technology Programming Environents Manual, Version 2.0. July 10, 2003.

[3] Freescale: AltiVec Technology Programming Environments Manual, Rev. 3. 04/2006.

[4] Freescale: AltiVec Technology Programming Interface Manual, Rev. 0. 06/1999.

[5] Johann Großschaedl: Instruction Set Extension for Long Integer Modulo Arithmetic on RISC-Based Smart Cards. Graz University of Technology, 2002.

[6] Johann Großschaedl, Erkay Savas: Instruction Set Extensions for Fast Arithmetic in Finite Fields. Graz University of Technology, 2004.

[7] Ian Ollmann, Ph.D.: AltiVec. 2003.